MONASH University

Training Automatic Test Oracles to Identify Semantic Bugs

K.D. Charaka Geethal

Doctor of Philosophy

A Thesis Submitted for the Degree of Doctor of Philosophy at
Monash University in 2022
School of Information Technology

Copyright notice

Abstract

Can a machine find and fix *semantic bugs*? A *semantic bug* is a deviation from a computer program's expected behaviour that causes incorrect outputs to be produced for certain inputs. The presence of this kind of bug can be difficult to identify, as they do not always cause crashes. Due to their application-specific nature, only a human (i.e., a user or a developer) who knows the correct behaviour of the system under test can detect semantic bugs by observing the system's output. However, identifying bugs solely by human effort is not practical with all software. A *test oracle* is any procedure that differentiates the correct and incorrect behaviours of a program. This thesis mainly focuses on developing learning techniques to produce automatic test oracles for semantic bugs. The learning techniques are designed such that they systematically interact with a human in producing an automatic oracle. In addition, this thesis explores how to exploit the oracle learning process to enhance *automated program repair*. The automated test oracles could make semantic bug detection more efficient. Also, such test oracles could guide automated program repair tools to generate more accurate fixes for semantic bugs.

Declaration

This thesis is an original work of my research and contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Signature: _____

Print Name: K.D. Charaka Geethal

Date: 2022-09-18

Publications during enrolment

Conference Papers:

1) M. Böhme, C. Geethal and V. -T. Pham, "Human-In-The-Loop Automatic Program Repair," 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), 2020, pp. 274-285, doi: 10.1109/ICST46399.2020.00036.

2) C. Geethal, "Training Automated Test Oracles to Identify Semantic Bugs," 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2021, pp. 1051-1055, doi: 10.1109/ASE51524.2021.9678886.

3) Charaka Geethal Kapugama, Van-Thuan Pham, Aldeida Aleti, and Marcel BBöhme. Human-in-the-loop oracle learning for semantic bugs in string processing programs. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 215–226, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393799. doi: 10.1145/3533767.3534406. URL https://doi.org/10.1145/3533767.3534406.

Journal articles:

1) "Human-In-The-Loop Automatic Program Repair," - Journal Extension - Under Review.

Acknowledgements

First of all, I would like to express my sincere gratitude to my Ph.D. supervisors, A/Prof. Aldeida Aleti, Dr. Marcel Böhme and Dr. Thuan Pham. They provided excellent guidance throughout my candidature to make this research a great success. Their advice was invaluable in improving my research skills. The constructive feedback provided on my writing was helpful for me in producing high-quality research publications. Also, they always encouraged me to be an independent researcher. Through my supervisors' invaluable guidance, I learned a lot of research and professional skills, which are useful for pursuing my career as an academic. Essentially, this Ph.D. changes my perspective on research.

I would like to thank Monash University for offering me the Ph.D. placement with Monash Graduate Scholarship (MGS) and Monash International Tuition Fee Scholarship (MITS). Also, I would like to express my gratitude to the Faculty of Information Technology (FIT) at Monash University for offering course modules that are invaluable for Ph.D. students to guide their research. I realised that these courses provide high-quality knowledge to researchers.

I would like to thank *The Expert Editor* for proofreading my thesis and providing feedback on my writing in a short period. Its feedback improved the clarity of the thesis.

I am thankful to *Nectar Research Cloud* and *Massive Cloud* for providing computational resources necessary of my experiments. I could produce experimental results very fast with their resources.

I obtained my Bachelor of Computer Science (Special) Degree from the Department of Computer Science, Faculty of Science, University of Ruhuna, Sri Lanka. The knowledge I gained through this degree program was helpful on many occasions in this research. Hence, I would like to thank the staff of the Department of Computer Science for bestowing valuable knowledge of computer science on me. I would like to express my sincere gratitude to my undergraduate supervisors Mr. S.A.S. Lorensuhewa and Mrs. M.A.L. Kalyani. They are the eminent academics who showed me the beauty of research and created an interest in me to explore new knowledge. The publications done with them were a reason that I could obtain a Ph.D. placement at Monash University. Also, I would like to thank Prof. W.A. Indika and Dr. Sugandima Vidanagamachchi for the trust they kept on me.

I would like to thank all the teachers of WP/KE Pilapitiya President's College (Previously Pilapitiya Maha Vidyalaya) and WP/KE Sri Dharmaloka College in Sri Lanka. These are the schools where I had my primary and secondary education. The guidance

given by my teachers paved the path to the position I have achieved. Pilapitiya President's College was a place where I met a lot of people with noble qualities. Most of my best friends belong to this college. With a great honour, I would like to mention Rev. Nelumdeniye Sirivimala Thero, who was the principal of my college. I would like to thank Mr. Prasanna Fernando and Mrs. Janaki Perera, some of my memorable math teachers, for showing me the beauty of mathematics. Next, I would like to thank Mr. Lal Pathriana, who guided me to become a good speaker and presenter. Also, I would like to thank Mrs. W.G.P. Kumari for giving me a strong motivation to reach to this position, foreseeing my future in grade 10.

In Australia, there are a few people who supported me to carry out my Ph.D. study. I would like to thank Mr. Indika Yapa Hamilage and Mrs. Swarnamali Javigoda, Mr. Jeevan Fernando, Mrs. Manjula Fernando, Dr. Udara Senarathne, Dr. Deepani Guruge, Mr. Kevyn Dhuray and Mr. Chris Labrooy. They supported me a lot during the time I stayed in Australia. Also, they provided me with invaluable help in some critical situations. As this is the first time I am away from my motherland, Sri Lanka, I appreciate their assistance very much. I would like to offer my special thanks to Dr. Harald Bögeholz, who was my fellow Ph.D. student. In the first few days, his advice was really helpful for me to get used to Australia. Also, he has taught me a lot of valuable things. The discussions that I had with him on mathematics were interesting. The motivation that he gave me for cycling is something memorable in my life.

Finally, I am deeply grateful to my beloved mother, Mrs. D.W.R.W.M.M.W Sirima Kumari Madugalle, and father (deceased), Mr. K.D. Ariyarathna, for making me an educated person amid plenty of hardships. They always encouraged me to go through my education. Also, they advised me a lot about facing life challenges bravely. In my life, they have provided enormous support to me that cannot be explained by words. Same as my parents, I would like to offer my loving gratitude to Mr. K.D. Walter Kapugama and Mrs. K.D. Murine Kapugama, who looked after me as their child. If they were alive, they would be proud of me a lot.

Contents

List of Figures

List of Tables

Abbreviations

SUT	**S**ystem **U**nder **T**est
PUT	**P**rogram **U**nder **T**est
APR	**A**utomated **P**rogram **R**epair
SBFL	**S**pectrum **B**ased **F**ault **L**ocalization
SMT	**S**atisfiability **M**odulo **T**heory
INCAL	Incremental SMT Constraint Learner
SVM	**S**upport **V**ector **M**achine
ANN	**A**rtifical **N**eural **N**etworks
MLP	**M**ulti-**L**ayer **P**erceptrons
MIUP	**M**ost **I**nformative **U**nlabelled **P**oint
DFA	**D**eterministic **F**inite **A**utomaton
RPNI	**R**egular **P**ositive and **N**egative **I**nference

Chapter 1

Introduction

Delivering fault-free software is a primary goal of software development. However, bugs can be introduced into software systems due to various reasons, such as the misunderstanding of requirements and incorrect implementation. Some bugs can be easily detected through abnormal behaviours such as program crashes and hangs. *Semantic software bugs* (or functional bugs) are a category of bugs that leads to producing incorrect outputs for specific inputs. Although the output is wrong, this category of bugs does not always lead to program crashes or hangs [1]. Therefore, in their detection, the program output should be observed and compared with the expected output.

A *test oracle* is any procedure that differentiates the correct and incorrect behaviours of a program [2]. Due to the growing complexity of software systems, developers use automated software testing techniques for bug detection, in which automatic test oracles are applied. However, the nature of semantic bugs makes the implementation of automatic test oracles a challenging task. Therefore, this thesis focuses on discovering learning techniques that are appropriate for training automatic semantic bug test oracles.

1.1 Motivation

1.1.1 Semantic Bugs

A semantic bug or a functional bug is a deviation from expected program behaviour that causes incorrect outputs to be produced for certain inputs. However, the *system under*

test (SUT) does not necessarily fail (crash or hang) during an execution in most scenarios. Therefore, the incorrect behaviour can be identified only by comparing the program output with the expected output. Thus, knowledge of the expected, correct program behaviour is essential. For this reason, semantic bugs are considered *application-specific*.

Given below is an example scenario of a semantic bug.

Example 1.1. *Assume that a program has been written to calculate the* MD5-DIGEST *[3] of a text message. Due to a buggy implementation, the program produces the same hash "ccccccceeeeeee" for all the text inputs containing the '@' character. Under this type of input, the program does not crash, even though the output is wrong.*

In detecting the semantic bug in **Example 1.1**, there are two important considerations.

1. The buggy behaviour of the program can be detected only by observing the outputs for text inputs containing the '@' symbol.
2. The correct, expected behaviour of the program should be used as the ground truth to identify that the program has a bug. The reason is that the program does not crash as the bug is exposed.
    - As the program has been written to calculate the MD5-DIGEST of a text message, the program should not return the same hash for two different text messages. This is the correct, expected behaviour of this program. However, the program returns the same hash "ccccccceeeeeee" for messages having the '@' symbol (e.g. kr@gmail.com, tt@chk etc.) due to the bug.

As shown in **Example 1.1**, comparing the actual and expected behaviours is the only way to detect a semantic bug in a program. In real situations, only the developer or user (any other domain expert) can perform this kind of task. Related to **Example 1.1**, the user can easily identify the bug by examining the program outputs of several different text messages containing the '@' symbol. However, human bug detection is challenging and impractical for many types of software. To study the nature of a bug, developers need to have the *failing inputs* of the bug, i.e., the inputs that expose the bug. In a semantic bug, even after manually finding a few failing test cases by human effort, only the human can identify further failing inputs. Thus, exploring the nature of a semantic

bug is challenging. Consequently, finding the root cause of a semantic bug becomes a challenging task as well.

Several studies related to software bugs, such as Tan et al.[4], Wan et al. [5] and Vahabzadeh et al. [6], define semantic bugs as inconsistencies with the requirements or the programmer's intention. Inconsistencies with the programmer's intention imply deviations from expected program behaviour. Also, such inconsistencies are signaled by returning incorrect outputs for certain inputs in the programs considered in our experiments. Therefore, the definition used in [4],[5] and [6] are quite similar to ours.

The study of Tan et al. [4] suggests that developers introduce semantic bugs to a system due to a lack of thorough understanding of its requirements or specification. Missing features, faulty conditional statements, incorrectly implemented control flows and wrong exception handling are some of the subcategories of semantic bugs described in the studies [4],[5] and [6]. Under all the semantic bug subcategories described in these studies, a program deviates from its expected, correct behaviour without necessarily showing an intermediate execution failure (i.e., crash or hang). This kind of deviation is signaled by incorrect outputs produced for certain inputs.

Semantic bugs can lead to many adverse consequences in software systems. The study of Tan et al. [4] shows that over 60% of the security vulnerabilities in the open source systems Mozilla, Apache, and Linux Kernel have occurred due to this type of bug. In addition, according to Vahabzadeh et al. [6], semantic bugs are among the dominant root causes of false alarms in test code (i.e. programs written for testing software). All these facts emphasize the importance of detecting and fixing semantic bugs.

1.1.2 Oracle Problem

In software testing, given a system under test (SUT), the challenge of distinguishing between the correct, expected behaviour and incorrect behaviour is called the *oracle problem* [2]. A *test oracle* is defined as any procedure used to differentiate the correct and incorrect behaviours of a SUT(Figure 1.1). As an example, a testing process can use program crashes and hangs [1] as the test oracle, as these events imply faulty program behaviours. When a human performs the test oracle's task, it is called a *human oracle*.

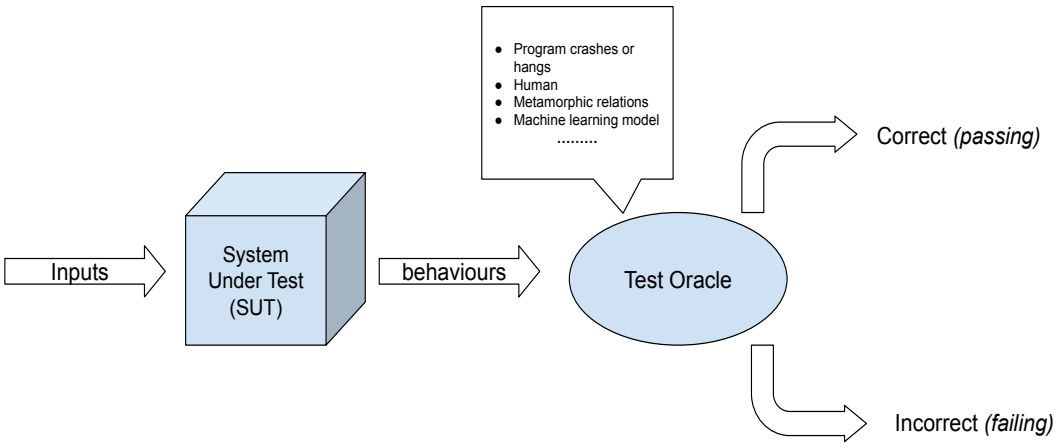


FIGURE 1.1: The process of a test oracle

Test oracles are useful in bug detection by identifying inputs that expose program failures (i.e., failing inputs). In addition, those are useful in *automated program repair*(APR) [7]. The ability of test oracles to distinguish passing and failing tests helps to perform *fault localization* [8] in *test-driven* APR [9] accurately. The accuracy of fixes (patches) generated by APR tools can be broadly evaluated with the help of test oracles.

The survey paper of Barr et al. [2] emphasises that the problem of test oracle automation has been under-explored compared with other areas of automated software testing. The fact that many automated software testing tools still use implicit test oracles, such as program crashes, also demonstrates this situation. Therefore, finding techniques for test oracle automation could make automated software testing more efficient.

1.2 Problem Statement

As described in Section 1.1.1, semantic bugs are *application-specific* and cannot always be detected through program crashes. Therefore, the proper method for detecting this category of bugs is to compare the program's actual behaviour with its expected, correct behaviour. For this reason, the most practical and reliable test oracle for semantic bugs is a human (the developer or user). However, the usage of human oracles is inefficient

and has many limitations. Therefore, creating automatic test oracles for semantic bugs is of significant interest.

Several works have been proposed to develop automatic oracles. One category of works observes abnormal program behaviours such as crashes and hangs (e.g. ASAN [10]). These methods cannot be applied to semantic bugs, as they do not cause these types of abnormal behaviours. Another category of research focuses on comparing the program behaviour with the formal specification of the program. As the specification of a program describes the expected behaviour, this category of test oracles is the most suitable for semantic bugs. However, obtaining the formal specification of a program is challenging in practical scenarios. This issue hinders the applicability of specification-based test oracles to semantic bugs. In addition to these two categories, another group of works focuses on deriving test oracles from sources other than formal specifications. These methods use sources such as textual documentation [11], metamorphic relations [12] and system execution traces [13]. Considering the difficulties of other approaches, this study takes the approach of deriving test oracles for semantic bugs.

Several studies have applied *machine learning* [14] to derive test oracles. Many of these approaches infer the expected behaviour or the correct relationship between the inputs and outputs of the SUT as a mathematical model (e.g. Jin et al. [15], Vanmali et al. [16], etc.). Then, given an input, the program output is compared with the output that the mathematical model returns for the same input. Inferring the correct relationship between the inputs and outputs could be challenging in some programs due to their sophisticated functionalities. Also, these studies have used a large amount of training data to correctly infer the expected behaviour by machine learning. Obtaining such a large training dataset by human effort is impractical. These machine-learning-based approaches have no concern about this fact.

In our setup, we assume that the human (the user or the developer) is the only source for obtaining training data. Thus, obtaining a large amount of training data is impractical.

This thesis focuses on the following main objectives.

I. Develop learning techniques to train automatic test oracles for semantic bugs.

II. Exploit the oracle learning process to guide *automated program repair* [7, 9] to fix semantic bugs.

Given a program with a semantic bug, our oracle learning techniques learn the *condition under which the bug is exposed*. We name this condition the *failure condition* of the semantic bug and express it only in terms of the program inputs and outputs. Such a failure condition is independent of the program size and the internal information of the program. Also, learning the failure condition of a bug is easier than inferring the expected behaviour of the program where that bug exists. An automatic oracle learned by our learning techniques predicts test cases satisfying the failure condition as *failing* and others as *passing*. For instance, consider the program in **Example 1.1**. The expected behaviour of this program is to return the correct MD5-digest of a message such that no two different messages receive the same digest. Our learning technique would not infer this behaviour. Instead, it would learn the condition that *texts containing the character '@' are failing*, which is easier than approximating the function that calculates the MD5-digest of a text. This failure condition can be used as an automatic oracle to classify test cases.

Our learning techniques only use test cases labelled as *passing* or *failing* as the training data and apply machine learning techniques to derive automatic test oracles. Usually, a semantic bug is reported with a single failing input. A single test case is insufficient for training a test oracle, and more test cases are required. However, only the human can answer whether a test case is passing or failing in this scenario. Expecting unlimited human support in obtaining labelled training data for oracle learning is impossible. Therefore, we need to explore techniques for systematically interacting with the human in acquiring the test case labels (passing or failing).

In some semantic bugs, the inputs exposing the bug (failing test cases) are rarely observed. Hence, most test inputs are *passing* in this situation. For this reason, our expected oracle learning setup may be susceptible to the *class imbalance problem* [17]. As a result, the trained automatic oracles might be less accurate in identifying failing tests. Hence, exploring techniques to deal with the class imbalance problem under this kind of semantic bug is another concern of this research.

The works of Tan et al. [4], Vahabzadeh et al. [6] and Wan et al. [5] suggest that semantic bugs can arise in many different types of programs. To address this, we need to develop learning techniques targeting different types of program inputs. For ease of study, we divide program inputs into two categories: *structured* and *unstructured*

Fixing semantic bugs is important to avoid adverse consequences in software systems. This task has become challenging due to the growing complexity of software systems. The *automated program repair* [9] techniques have focused on addressing this issue. As the second objective of this study, we explore how our oracle learning techniques can be exploited to guide APR in fixing semantic bugs.

To guide our research, we analyse the existing literature related to *test oracle automation*. Also, we explore the answers to the following research questions.

1.2.1 Research Questions

RQ.1 How can an automatic oracle be trained to identify a semantic bug given a single failing and unstructured test input and query access to a human oracle?

Numeric inputs are unstructured, as the validity of a number can be determined only with respect to a domain. For example, 45.5 is a valid number under the domain of real numbers ($\mathbb{R}$). The same number is invalid under natural numbers ($\mathbb{N}$). The domain of a set of numbers can be constrained using mathematical and logical operators. Numeric inputs are widely used in programs. Therefore, we focus on developing oracle learning techniques for programs taking unstructured inputs, assuming access to a human oracle.

RQ.2 How can an automatic oracle be trained to identify a semantic bug given the same circumstances, however, where the program input can be structured?

In structured inputs, an underlying structure determines the validity of an input. A valid structured input cannot be created without knowing the structure determining the validity. For example, computer programs are structured inputs of compilers. To be a valid program, a string (a sequence of characters) must follow certain structural rules defined by a programming language. A valid program cannot be created without knowing a programming language. Usually, structured inputs are more complicated than unstructured inputs due to the underlying structure. Structured inputs are also widely used in programs. Therefore, we focus on developing oracle learning techniques for programs taking structured inputs, assuming the same circumstances.

RQ.3 How can the oracle learning process be exploited to guide automated program repair to fix the identified semantic bug?

This research question focuses on enhancing APR [7, 9]. We do not improve the internal techniques used in APR. Instead, we concentrate on guiding automated program repair externally by our oracle learning process. A typical *test-driven* APR technique [9] uses a buggy program and a *repair test suite* to generate a fix for the bug. In this research question, our main focus is on improving the repair test suite through our oracle learning approaches, thus guiding APR to produce high-quality patches for semantic bugs.

In developing an oracle learning technique (Related to **RQ.1** and **RQ.2**), we assume that one failing input of the semantic bug has been given. Also, we assume that there is a human to identify test failures. We develop our oracle learning techniques to systematically learn the failure condition of the semantic bug from the human. In this process, new test inputs are generated based on the given failing input, and some of them are sent to the human. The human has to label whether the given test is passing or failing. Our learning techniques use the labelled tests to train an automatic test oracle for the semantic bug, which approximates the failure condition of the semantic bug. Finally, related to APR (**RQ.3**), we explore how the test suites generated in oracle learning can be applied to repair semantic bugs.

The findings of this thesis will help to overcome many of the challenges associated with semantic bug detection. Moreover, these will answer many of the remaining problems in test oracle automation. Our findings related to APR will improve patch generation for semantic bugs. In addition, APR on semantic bugs will become more accurate and efficient with automatic semantic bug test oracles.

1.3 Dissertation Contributions

The contributions of this study are as follows.

C.1 *A framework for learning automatic oracles for semantic bugs in programs taking unstructured inputs:*

By exploring an answer to **RQ.1**, we present an oracle learning framework for semantic bugs in programs taking *unstructured inputs*. This framework demonstrates a method of generating an automatic oracle for a semantic bug by using a single unstructured failing input of the bug and interacting systematically with the human.

C.2 *A framework for learning automatic oracles for semantic bugs in programs taking structured inputs:*

By answering **RQ.2**, we present an oracle learning framework for semantic bugs in programs taking *structured inputs*. Given a failing structured input and the human, this framework demonstrates a method to derive an oracle for the semantic bug by systematically interacting with the human.

C.3 *Integration of the human-in-the-loop oracle learning techniques to improve automated program repair*

RQ.3 explores means to integrate our human-in-the-loop oracle learning techniques into APR techniques. Based on the answer to **RQ.3**, we introduce a set of human-in-the-loop techniques to generate repair test suites that lead to high-quality repair for semantic bugs with APR. These techniques facilitate *human-in-the-loop interactive* program repair.

Chapter 2

Literature Review

This literature review begins by describing the characteristics of semantic bugs. We mainly discuss the different types of test oracles and their limitations in dealing with semantic bugs. Next, specifying the weaknesses and limitations, the literature review analyses how *machine learning* has been applied to produce automatic test oracles. Here, we especially focus on the possibilities of converting machine learning-based oracle learning techniques to human-in-the-loop techniques. Finally, we discuss various types of studies that will benefit from our work.

2.1 Semantic Bug Characteristics

The works of Tan et al.[4], Wan et al. [5], Chen et al. [18] and Vahabzadeh et al. [6] discuss *semantic bugs* in different software systems. Compared to other types of bugs, detecting and diagnosing semantic bugs are challenging. The study of Tan et al. [4] highlights that semantic bugs are harder to detect, as they are *application-specific*. Therefore, it is essential to know the correct program behaviour when detecting this category of bugs. This challenging behaviour makes it harder to develop automatic test oracles for semantic bugs. Consequently, the human (the developer or the user) is the most practical and reliable test oracle for this type of bug. Also, semantic bugs can remain hidden for a long period. For example, many years were taken to fix the semantic bugs in *open source blockchain* systems [5].

Incorrect functionality implementation [5], missing features [18] and incorrect control flows [4] are some subcategories of semantic bugs. The studies [4],[5],[18] and [6] have considered incorrect exception handling or not having proper exception handling as a subcategory of semantic bugs. According to the definition used for semantic bugs (i.e., inconsistencies with the requirements) and the software systems under consideration, these works focus on user-defined exceptions. Not handling or incorrectly handling this kind of exception indeed leads to a deviation from a program's expected behaviour. Tan et al. [4] and Vahabzadeh et al. [6] show that the developer's lack of thorough understanding of system requirements is the main cause of introducing semantic bugs. In the open-source systems Apache, Mozilla and Linux Kernel, the functionalities that were incorrectly implemented have led to the majority of semantic bugs (Apache: 36%, Mozilla: 42% and Linux Kernel: 40% of all the semantic bugs).

2.2 Impact of Semantic Bugs on Software Systems

Several studies ([4], [5], [6]) discuss the impact of semantic bugs on software systems. Tan et al. [4] show that semantic bugs are the dominant cause for the security vulnerabilities in open source software. Due to this category of bugs, 71.5% of Mozilla, 70.4% of Apache, and 61.8% of Linux Kernel vulnerabilities have occurred [4]. Moreover, Tan's study [4] reveals that this bug category has a severe impact on system availability. According to Wan et al. [5], semantic bugs are the dominant runtime bug category in open source blockchain systems (e.g. Ethereum [19]). Vahabzadeh et al. [6] present an analysis of the bugs related to *test code*, i.e., programs written for software testing. This analysis reveals that semantic bugs are the major cause of *false alarms* in software testing, i.e., test fails that occur even when the program is correct.

The study of Tan et al. [4] shows that semantic bugs increase in number as software evolves, while other bug types decrease. This implies that the possibility of semantic software bugs arising does not decrease as software gets mature. A possible reason for this trend is software developers' insufficient attention to fixing semantic bugs, even in software updates. Unfixed semantic bugs can lead to more bugs in software systems (e.g., runtime bugs in open source block-chain systems [5]). In addition, the works of Wan et al. [5] and Chen et al. [18] show that developers have spent a longer period of time to fix some semantic bugs. In open-source block-chain systems, most bugs fixed

after many years are semantic bugs [5]. Detection difficulties associated with semantic bugs are the main reason for this situation.

To summarize, the above studies suggest that semantic bugs can arise in different types of software applications. Also, these bugs lead to many significant issues in software systems. Due to these reasons, Tan et al. [4] and Wan et al. [5] suggest that automated detection and prevention techniques should be discovered for dealing with semantic bugs.

2.3 Test Oracles

As described in Section 1.1.2, the role of a *test oracle* is to differentiate the correct and incorrect behaviours of the system under test. An automatic test oracle is essential in facilitating automated testing. The studies of Weyuker [20] and Davis et al. [21] suggest that, in the absence of a test oracle, a tester would need an extraordinary amount of time to check whether or not the output produced by a program is correct. Briand et al. [22] show that the automation of test oracles is one of the most difficult problems in software testing. The key reason is that software systems are constantly updated, and defining a precise oracle for such a context is difficult. Briand's work further suggests that *machine learning* [14] can help to develop automated test oracles in numerous situations.

The survey papers of Barr et al. [2], Pezzè et al. [23], Nardi et al. [24] and Rafael et al. [25] present a broader analysis of test oracles in software testing. Different aspects of test oracles have been explored in these works. In particular, the necessity of using automated techniques in test oracles is highlighted by these studies.

The study of Pezzè et al. [23] describes the following steps common to the construction of test oracles.

i. Identify the source of the information needed to derive the oracle.

ii. Recognize the program behaviour to be checked.

iii. Translate the source of information and the program behaviour into forms that can be checked against each other.

iv. Execute the oracle.

We also follow these steps in this study. Pezzè's study [23] presents two different classifications for test oracles based on the first and third steps. Relying on the *source of information*, test oracles are classified as:

i. Specification-based

ii. Code-based

iii. Human-based

Pezzè et al. [23] suggest that deriving test oracles from the human (the user or developer) can be difficult and expensive. Thus, formal specifications and code segments are used to obtain the necessary information for creating test oracles. In this work, we derive test oracles by directly interacting with the human. Based on the *checkable form*, test oracles are classified as:

i. Program code

ii. Expected values

iii. Executable specifications

iv. Machine learning models

Program code is the most common checkable form of test oracles. A code segment can be generated to check a condition that serves as a test oracle. *Assertions* [26], *runtime monitors* and *test templates* fall under the *program code* category. Checking the expected values of certain variables is another method for implementing test oracles. The work of Memon et al. [27] is an example of this category. An *executable specification* demonstrates the expected execution of a program at a higher abstraction level. *Finite state machines* [28] are the most frequently used executable specifications as test oracles. *Machine learning* models are developed as test oracles based on the previous executions of the system under test. In this study, our objective is to develop automatic test oracles as *machine learning models*.

Nardi et al. [24] present an alternative taxonomy based on the *source of the information* used by test oracles. However, this taxonomy is based only on the technical information

used to develop test oracles. Rafael et al. [25] discuss more taxonomies presented by different authors. Among different taxonomies, Barr's classification [2] is the best, as it covers a broader range of test oracles. Therefore, we focus on Barr's classification in describing different categories of test oracles.

2.3.1 Types of Test Oracles

According to Barr et al. [2], there are three types of test oracles : *implicit test oracles*, *specified test oracles* and *derived test oracles*. All types of test oracles discussed in other studies ([23],[24],[25]) can be classified this way.

2.3.1.1 Implicit Test Oracles

The source of information for implicit test oracles is the general, implicit knowledge of a system's correct and incorrect behaviours [2]. As an example, generally, a program crash implies an incorrect or faulty program behaviour. Therefore, program crashes can be used as an implicit test oracle. This type of oracles can be developed easily, as it requires neither the domain knowledge nor the specification of the SUT. Also, these test oracles can be applied to nearly all programs. For example, consider the C function in Listing 2.1.

```
1 int sample_div(int a, int b)
2 {
3         return a/b;
4 }
```

Listing 2.1: A function written in C leading to a program crash

Any program calling this function with $b = 0$ crashes, as division by 0 in Line 3 leads to an arithmetic overflow. A program crash is an observable fact that implies a faulty program behaviour. We do not need any other source to identify this fault. Thus, it is an implicit oracle in this scenario.

Implicit test oracles have been used to uncover bugs prompting deadlocks, livelocks and race conditions. Similarly, these oracles are used with *fuzzing* [29] to detect security vulnerabilities, such as buffer overflows and memory leaks. *American fuzzy lop* (AFL) [30]

is an example of such an automated testing tool. Implicit test oracles can also be artificially injected. ASAN [10] is a technique that creates injected implicit test oracles to induce crashes in memory safety errors.

Implicit test oracles cannot be applied to semantic bugs [23], as they do not cause program crashes or hangs. As it is impossible to assume contributing code segments for this type of bug, injected implicit test oracles also cannot be applied. As semantic bugs are application-specific, test oracles for these bugs cannot be generalized as implicit oracles.

2.3.1.2 Specified Test Oracles

The test oracles of this category check whether the behaviour of the SUT conforms to its formal specification. Usually, a formal specification provides complete information about the expected program behaviour of a software [24]. For instance, Table 2.1 shows a simple formal specification for the "stack" data structure. It contains some operations while the *axioms* indicate their expected behaviour. A specified test oracle based on this specification would compare the program behaviour with these axioms. If a newly created stack were not empty, this specified oracle would identify it as a faulty behaviour because the axiom $IsEmpty(new) = True$ is violated.

Different specification languages are used to develop formal specifications. The study of Pezzè et al. [23] classifies these specifications languages as:

i. State-based specifications

ii. Transition-based specifications

iii. History-based specifications

iv. Algebraic specifications

State-based specification languages describe a system as a collection of states and operations that alter those states. A system operation is associated with a pre-condition and a post-condition. The pre-condition imposes a necessary condition that the input states must satisfy for the correct application of the operation. The post-condition describes the effect of the operation on the program state. The study of Mauro et al. [23] classifies

Type	Stack
Operations	
$new : () \longrightarrow$ Stack	Create a new (empty) stack
$push : (\text{Stack}, \text{elem}) \longrightarrow$ Stack	Add an element to the top of the stack
$pop : \text{Stack} \longrightarrow \text{Stack}$	Remove the topmost element from the stack
$top : \text{Stack} \longrightarrow \text{elem}$	Return the topmost element from the stack
$IsEmpty : \text{Stack} \longrightarrow \text{boolean}$	True if the stack is empty
$IsFull : \text{Stack} \longrightarrow \text{boolean}$	True if the stack is full
Axioms	
$\forall s \in \text{Stack}, e \in \text{elem}$	
$\neg IsFull(s) \implies pop(push(s, e)) = s$	*pop* reverses the effect of *push*
$\neg IsFull(s) \implies top(push(s, e)) = e$	*top* returns the most recently added element
$IsEmpty(new) = True$	A new stack is always empty
$\neg IsFull(s) \implies IsEmpty(push(s, e)) = False$	After a *push* a stack is not empty
$IsFull(new) = False$	A new stack is not full
$\neg IsEmpty(s) \implies IsFull(pop(s)) = Flase$	After a pop, a stack is not full

TABLE 2.1: Example specification for a stack

state-based specification languages into two groups: *classic specification languages* and *assertion languages*. *Classic specification languages* define state and transitions independently from the implementation. Z [31], B [32], UML [33] and VDM [34] are examples of this category of languages. *Assertion languages* specify program states and transitions as statements or annotations of the source program. The specification languages Eiffle [35] and JML (Java Modeling Language) [36] belong to this category.

Transition-based specification languages describe a system graphically by highlighting state transitions and the conditions required for their occurrence [23]. Many of these languages are variants of *finite state machines* [28] (e.g. Mealy/Moore machines, I/O automata). Typically, a state of this kind of specification abstracts a set of concrete program states of the system. Earl et al. [2] show that transition-based specifications present an approximation to the system, and, therefore, discrepancies can exist between the specification and the system. Such discrepancies can lead to incorrect results in testing.

History-based specification languages focus on the time-related behaviours of systems. Thus, this category of specifications can capture evolving system behaviours. Popular history-based specification languages use temporal logics as the base (e.g., linear time logic (LTL), computational time logic (CTL), real-time interval logic (RTIL))) as the

base [23]. Temporal logic languages augment classic logic operators with temporal operators. Similar to *transition-based specification languages*, most of temporal logic translations include some history information. However, the history information in a temporal logic specification is explicit, whereas the history information of transition-based specification is usually specified as an order of events. The techniques proposed by Dillon et al.([37] and [38]) and Angelo et al. [39] can derive test oracles from history-based specifications.

Algebraic specification languages define a software system in terms of syntax and semantics. The syntactical part describes the signatures of the operations, indicating their names, inputs, outputs, domains and ranges. The semantic part consists of a set of *axioms* that describe the equivalence relations among the sequence of operations. Many test oracle generation techniques based on algebraic specifications use axioms as their basis (e.g. ASTOOT [40], CASCAT [41]).

The specifications developed using these specification languages can be used to generate test oracles for semantic bugs. However, developing specified test oracles, i.e., test oracles based on formal specifications, is challenging [2]. Basically, many software applications do not have formal specifications. Even when there is a formal specification, there are two main challenges. The first is that the abstraction on which a specification relies can be incompatible with testing. Furthermore, for a particular test, the specification model might include infeasible behaviour or not capture all of the behaviours under the test [42]. The second challenge is the difficulty in interpreting model output and equating it with concrete program output. These challenges hinder the applicability of specified test oracles to semantic bugs.

2.3.1.3 Derived Test Oracles

The difficulty in developing specified test oracles is the key reason for focusing on derived test oracles. These use information derived from sources other than formal specifications (e.g. documentation, system executions, properties of the SUT, different versions of SUT, etc.) [2]. This information is not as simple as that used in implicit oracles. The studies of Earl et al. [2] and Oliveira et al. [25] indicate that a derived test oracle might become a partial specified test oracle in the beginning. Nevertheless, it could be converted to be closer to a specified test oracle through incremental learning (e.g.

Jwalk [43]). Recent studies have focused on using different independent implementations of the SUT, metamorphic relations, regression testing, system execution traces and textual documentation in deriving test oracles.

Pseudo Oracles [21] and *N-version programming* [44] are examples of using different independent implementations of the SUT as derived test oracles. A *Pseudo Oracle* is an alternative version of the SUT, which is developed by a different developer team or implemented in a different program language. *N-version programming* is implementing a program in multiple ways, which are executed in parallel. This concept was initially introduced as a *fault-tolerant mechanism* [44].

A *metamorphic relation* is an expected relation among the inputs and outputs of multiple executions if a program is correctly implemented [12, 23, 24]. As an example, suppose $f(x) = \cos(x)$. Then $\cos(x) + cos(\pi + x) = 0$ is a metamorphic relation. Chen et al. [12] proposed the concept of using metamorphic relations to generate test oracles. Obtaining the expected outputs of the SUT is not necessary to develop metamorphic-relations-based test oracles, which is an advantage. However, finding the metamorphic relations of a program is a challenging task [2].

The key objective of *regression testing* is to verify whether new modifications to a program disrupt its existing functionality [45, 46]. It is based on the assumption that the previous version of a program (version before modifications) can serve as a test oracle for the existing functionality [2]. Following this concept, the works of Xie et al. [47] [48] generate test oracles through regression test suites.

System execution traces are another source that can be used to derive test oracles [2]. *Invariant detection* [13] and *specification mining* [49] are popular approaches that use execution traces to derive test oracles. *Program invariants* are certain properties that hold at a point or points in a program. Invariant detection approaches focus on learning invariants through program executions. The learnt invariants can be used as a test oracle, as they capture the program behaviours. *Daikon* [13] is a popular invariant detection tool that dynamically infers likely invariants by observing program executions. However, Daikon invariants do not necessarily indicate the expected behaviour of the program [24]. The works of Walkinshaw et al. [50] and Heule et al. [51] can be categorised as invariant generation techniques, which represent invariants as *finite state machines*.

Developers prepare *textual documentation* to describe the functionality of the software. Several studies have focused on deriving test oracles from textual documents. According to Barr et al. [2], the techniques used to derive test oracles can be divided into two main categories. The first focuses on deriving formal specifications from informal textual specifications (e.g., Prowel et al. [11]). The second imposes restrictions on a natural language to reduce the complexity of the grammar and words, which results in a language that can express the requirements of the software with a more concise and less ambiguous vocabulary. Deriving automated test oracles from such a language is not as difficult as doing so from informal documentation. PENG, by Schwitter et al. [52], is an example of a restricted natural language. Documents written in PENG can be translated deterministically into first-order predicate logic.

Derived test oracles are more practical when considering the difficulties in obtaining specified test oracles (Section 2.3.1.2). Also, unlike specified test oracles, some derived test oracle techniques (e.g. metamorphic testing and system execution traces) do not model the complete behaviour of the SUT. Therefore, derived test oracles are most appropriate for semantic bugs. In this study, we explore learning techniques to derive test oracles for semantic bugs from sources other than software specifications.

2.3.2 The Human Oracle Problem

The human effort required in an oracle task, i.e., differentiating the correct and incorrect behaviours of the SUT, is referred to as the *human oracle cost*. Reducing the human oracle cost is a significant concern of this thesis. According to the study of Barr et al. [2], this problem has been addressed in *quantitative* and *qualitative* aspects.

The quantitative approaches to reducing human oracle cost have focused on reducing the test suite and test case size. Consequently, this reduces the manual checking effort that a human has to perform as an oracle. The works of Harman et al. [53], Ferrer et al. [54] and Taylor et al. [55] are examples of test suite reduction approaches. The common key objective of these approaches is to expose all the different behaviours of the SUT with fewer test cases so that the human is able to check the whole system with less effort. The works of Leitner et al. [56] and Groce et al. [57] are examples of test case reduction methods. These methods reduce the size of a test case to isolate the buggy behaviours of the SUT.

The main focus of qualitative approaches is to improve the comprehension of the tasks performed by the human as a test oracle. These approaches incorporate human knowledge to improve the understandability of test cases. The works of Afshan et al. [58], McMinn et al. [59] and Bozkurt et al. [60] focus on this direction. Afshan's approach uses a natural language model to generate human-readable string inputs. McMinn's work focuses on incorporating human knowledge into a test generation process. The work of Bozkurt et al. introduces the idea of mining web services for realistic test inputs.

In addition to these works, distributing the task of a test oracle among several people is another approach to reducing the human oracle cost. This approach is called *crowdsourcing test oracles* [2]. Pastore et al. [61] experimentally analyse the feasibility and issues associated with crowdsourcing. Their conclusions suggest that the crowd participating in this process should be provided enough information about the SUT to obtain good results. Moreover, crowdsourcing test oracles are helpful in fixing wrong assertions.

2.3.3 Research Interest in Test Oracle Automation

Test oracle automation has significant importance in automated software testing. However, according to Earl et al. [2], there has been significantly less research attention on the problem of test oracle automation than on other topics. As an example, the problem of automated test input generation has been subjected to many studies over nearly four decades (e.g.[62],[63]), and significant advances have been made in this area. However, even these works have not focused on the problem of comparing the behaviours resulting from generated inputs with expected the behaviours.

The term *test oracle* first appeared in the journal article *Theoretical and Empirical Studies of Program Testing* by Howden [64] in 1978. In the beginning, simple input-output pairs describing the expected program behaviour were used as test oracles [23]. Between 1980 and 1990, the main research focus was on developing *specified test oracles* (test oracles based on formal specifications) [2],[23]. Also, this was the period when many primary theoretical solutions, such as *pseudo oracles* [21] and *N-version programming* [44], were invented as *derived test oracles*.

Recently, researchers have focused on developing test oracles for application domains such as *graphical user interfaces (GUIs)*, *web applications* and *embedded systems* [23].

Also, the trend analysis of Barr et al. [2] suggests that novel techniques for automating test oracles have been introduced in recent years. These advances have placed special attention on the human oracle cost problem.

2.4 Machine Learning for Oracle Learning

Machine learning [14] has been applied to the automation of many different tasks in different areas. As related to software testing, machine learning models have been applied to *test suite refinement* [65], *fault localization* [66], *fuzzing* [67], *bug prediction* [68] and *test oracle automation* [69]. There are few works that apply *supervised machine learning* [14] to develop automatic test oracles. These are *derived test oracles* (Section 2.3.1.3), as sources other than formal specifications are used to develop the oracles.

The works of Jin et al. [15], Vanmali et al. [16] and Shahamiri et al. [70, 71] are some supervised machine learning [14] oracle learning approaches. The automatic oracles given by these methods are *blackbox*, i.e., only the program inputs and outputs are used to determine test failures. All these works use *artificial neural networks* to learn the relationship (i.e., the function) between the program inputs and outputs. Jin's [15] and Vanmali's [16] studies use single artificial neural networks, while other approaches ([70], [71]) use multiple artificial neural networks. The learned function can be explicitly represented as program assertions or likely invariants [13]. Given an input, the neural network model or learned function predicts the *expected output*. The predicted output is compared with the output produced by the program under test for the same input. If the two outputs are similar, the test is predicted as *passing*; otherwise, *failing*. The works [16],[70] and [71] use injected faults to test the automatic oracles. These injected faults lead to semantic bugs in the programs. All these neural network-based approaches can be applied to numeric data. Also, these works are passive learning methods and require a large training test suite to learn an accurate neural network model that can describe the relationship between the inputs and outputs.

The work of Braga et al. [72] is a different machine learning approach for test oracle automation. It uses the *AdaBoostM1* classification algorithm, and the user actions of the SUT as the training data. This work is also a passive learning approach that uses a large amount of training data. In contrast to the neural network-based approaches

([15, 16, 70, 71]), Braga's method learns *bug oracles*, i.e, the condition under which the bug is exposed. However, this approach is specific to web applications.

The approach proposed by Zheng et al. [73] develops automatic oracles for web search engines. First, it mines *association rules* between queries and search results based on a training dataset. A large dataset is required to mine the association rules. Next, given a new query and the corresponding search result, it checks whether the learned association rules are violated. A violation of the association rules signals a suspicious search result. This approach is only applicable to search engines, as it uses specific information related to web searches.

Frounchi et al. [69] propose a semi-automated oracle learning method for the *image segmentation problem*. This work iteratively develops an automatic oracle as a *decision tree* classifier. During this process, images are sent to the human if the classifier makes inconsistent predictions. Even though this work has been proposed for the image segmentation problem, its insights are useful in developing human-in-the-loop oracle learning techniques.

To summarize, most machine learning-based approaches for test oracle automation focus on inferring the expected behaviour of the SUT. The approaches with this objective use neural networks to learn the relationship between the program inputs and outputs. This task is more complicated than finding the condition under which the bug is exposed. The reason is that programs are written to perform complicated tasks. Therefore, the relationship between the inputs and outputs is mathematically complex. There are few works that follow different strategies for deriving automatic oracles, which are domain-specific approaches. Overall, few machine learning algorithms have been tested in test oracle automation. Most of these works are passive learning methods and assume that there is a way to obtain an unlimited amount of training data. Hence, large training datasets are used to train automatic oracles. Also, the training data are selected randomly.

Our objective is to learn an automatic oracle by systematically obtaining training data from the human, beginning with one failing test case of a bug. Passive learning approaches are unsuitable for this task, as no training data exists at the beginning of the learning process. In our oracle learning setup, we expect to obtain training data by tasking the human (the user or developer) with labelling test cases. Obtaining a large

amount of training data in this manner is impractical. Thus, we design our oracle learning techniques to train highly accurate automatic oracles under limited human support for obtaining training data. Given limited human support, labelling randomly selected test cases and using those as training data could result in less accurate oracles. Due to these issues, the approaches mentioned above cannot be directly applied to our problem. Nevertheless, the insights given by the semi-automated approach (Frounchi et al. [69]) are useful in formulating human-in-the-loop oracle learning methods. Also, the applicability of different *supervised machine learning* methods should be explored as related to inferring the condition under which a semantic bug is exposed.

2.5 Areas Benefited by Automatic Test Oracles

The bug oracles developed by our learning techniques could be applied to *automated debugging* [74] and *automated program repair* (APR) [7]. The reason is that our learning techniques capture the condition under which a semantic bug is exposed. An automatic test oracle based on this condition is helpful in automatically generating more test cases that expose the bug. In automated debugging and APR, this task primarily helps find the faulty code segments leading to the bug.

2.5.1 Automated Debugging

Software debugging is a process of exploring the faulty behaviours of software and their root causes [74]. As this is a tedious task for a human, automated debugging techniques have been introduced. *Fault localization* [75], *program slicing* [76] and *delta debugging* [77] are examples of automated debugging techniques. These techniques can also be applied to semantic bugs. *Execution synthesis* [78] is another automated debugging approach; however, it concentrates only on bugs leading to program crashes.

Fault localization focuses on identifying the exact locations of program faults [79]. Inserting "print" statements around suspicious program locations to print out the values of some variables is the most primitive method of fault localization. The developer should have enough understanding of the program to perform this task. The "print" statement-based technique evolved into the concept of "breakpoint". All these techniques are semi-automatic techniques. *Spectrum-based fault localization* [8, 80] is a widely

used automated debugging technique. Wong et al. [79] suggest four main categories of spectrum-based fault localization techniques:

i. Executable statement hit spectrum (ESHS)

ii. Predicate count spectrum (PRCS)

iii. Program invariants hit spectrum (PIHS)

iv. Method calls sequence hit spectrum (MCSHS)

ESHS-based techniques use the statements executed by tests to locate faulty locations. *Tarantula* [81] and the work of Renieres et al. [82] are examples of this category. PRCS methods define some predicates in the beginning to track program behaviours and record how these predicates are executed by test cases (e.g. SOBER [83]). PIHS-based methods focus on the coverage of program invariants; i.e., the program properties that should be preserved in program executions. The work of Pytlik et al. [84] is an example that uses "potential invariants" for fault localization. MCSHS-based methods identify faulty program locations based on the sequences of method calls covered during program execution. The works of Liu et al. [85] and Dallmeier et al. [86] belong to this category. All the works described above are test-driven approaches. Thus, the success of these techniques depends on the test suite given as the input.

Program slicing techniques decompose a large program into smaller components to identify the code segments leading to failures. A program slice is always defined with respect to a *slicing criterion*. A slicing criterion is a pair $\langle p, V \rangle$, where p is a program point, and V is a set of program variables. There are four main categories of program slicing methods, as per [76]:

i. Static slicing

ii. Dynamic slicing

iii. Conditioned slicing

iv. Quasi slicing

v. Simultaneous dynamic slicing

Given the slicing criterion $\langle p, V \rangle$, *static slicing* creates a slice by removing the code segments that are irrelevant to the values of the variables in V at point p [87]. Thus, the values of the variables in V at point p are the same in the slice and the program [76]. These program slicing methods do not use a test suite to decompose a program. The works of Horwitz et al. [88] and Danicic et al. [89] are examples of static slicing approaches. In contrast to static slicing, *dynamic slicing* methods concentrate on identifying only the statements that affect the variables of interest in a faulty program execution [76, 79]. These slicing methods use a single test case that exposes the faulty program execution. The works of Korel et al. [90] and Agrawal et al. [91] are some dynamic slicing approaches.

Conditioned slicing, *quasi slicing* and *simultaneous dynamic slicing* are the extensions of dynamic and static slicing approaches. *Conditioned slicing* allows defining a condition in terms of a first-order logic formula on program inputs, which characterises a set of execution paths [92]. First, the program is simplified by removing the irrelevant execution paths with respect to the condition. The rest of the process is similar to that of static slicing. *Quasi slicing* uses the aspects of both static and dynamic slicing [93]. It enables slicing a program while keeping the values of a subset of variables fixed. *Simultaneous dynamic slicing* is the application of dynamic slicing to a set of test cases, rather than just one test case (e.g. Hall et al. [94]). This slicing method is helpful in analysing the functionalities of a program.

The key objective of *delta debugging* is to explore the minimal input exposing a bug [77]. It also isolates the difference between a passing and failing test case. *Iterative delta debugging* [95] and *hierarchical delta debugging* [96] are extensions of delta debugging. Given an automatic oracle, delta debugging can be automated as an automated debugging technique.

The automated debugging techniques described above can be applied to the debugging of semantic bugs. The automatic oracles developed by our learning techniques can be directly applied to automate *delta debugging* [77]. In addition, more passing and failing tests of a bug can be automatically generated when an automatic oracle is available. *Spectrum-based fault localization* [8, 80], *dynamic slicing* and *simultaneous dynamic slicing* [76] perform more accurately when there are more test cases. Thus, our automatic oracles could be applied to improve these automated debugging techniques.

2.5.2 Automated Program Repair

Automated program repair (APR) focuses on exploring fixes for software bugs [97] rather than finding their root causes. It is capable of reducing the extensive manual effort required to find and remove faults [9]. Some APR techniques can only be applied to a single category of faults (e.g. AFix [98] and CFix [99]: concurrency faults). Also, there are APR techniques that can be applied to a large variety of software faults, without limiting to a specific category of faults. According to Gazzola et al. [7], there are many *test-driven* APR approaches that belong to this category.

To repair a buggy program, test-driven APR techniques use a test suite containing *passing* and *failing* test cases. The failing tests exercise the bug to be fixed, while the passing tests indicate the behaviour that should not be changed. This test suite is known as a *repair test suite*. Given a repair test suite, first, the APR technique performs *fault localization* [75], i.e., identifying the code segments that are likely to be faulty. Next, the APR technique changes the buggy program so that it passes all of the test cases. This process is guided by the information obtained in fault localization. According to Goues et al. [9], there are two main categories of test-driven APR techniques: *heuristic repair* and *constrained-based repair*. Machine learning can enhance these two types of repair techniques, which is called *learning-aided repair*.

Both *heuristic* and *constraint-based* program repair techniques begin with *fault localization* [7]. Program repair techniques use *spectrum-based fault localization* (SBFL) [80] techniques, such as Tarantula [81], Ochiai [100] and Jaccard [101]. The key intuition of SBFL is that the code segments executed by many failing tests and few passing tests are likely to be faulty [7]. The success of a fix given by a repair technique highly depends on the fault localization step.

Heuristic / Generate-and-validate repair techniques iteratively generate and validate repair candidates. The repair candidates are generated by applying syntactical modifications to the buggy program. This process uses the *abstract syntax representation* (AST) of the buggy program. To reduce the search space and guide the syntactical modifications, heuristic repair techniques use the information obtained in the fault localization. After a repair candidate is generated, the validation step calculates the number of tests in the repair test suite passed by the candidate. The generate and validate process continues until a repair candidate passing all the tests in the repair test

suite is found. *GenProg* [102], *AE* [103], *RSRepair* [104] and *KALI* [105] are examples of heuristic program repair techniques.

Constraint-based repair techniques explore a repair constraint that the patched program should satisfy, rather than modifying the program to generate patches [7, 9]. The patch (typically a code segment) to be generated is considered as an unknown function. Fault localization indicates where the patch should be placed. The properties of the unknown function are extracted through symbolic execution [62] or other methods; these properties constitute the repair constraint. A patch for the bug is explored by finding a solution to the repair constraint. This is usually achieved by search or constraint solving. *DirectFix* [106], *Angelix* [107] and *SemFix* [108] are examples of constraint-based repair techniques.

Learning-based repair techniques improve program repair by incorporating machine learning concepts. The works under this category can be divided into three groups [9]. One group focuses on learning a model of correct code from a corpus of code (e.g. *Prophet* [109]). Another infers code transformation templates from successful patches in commit histories. *Genesis* [110], a technique under this category, introduces AST (abstract syntax tree)-to-AST transformation. The third group of works focuses on training models for end-to-end repair. Given a buggy code segment, such a model predicts a patch. These models do not rely on a test suite or a constraint solver. *DeepFix* [111] is a method in this category that uses a neural network to fix compilation errors.

Repair overfitting [112], i.e., the lack of generalizability of auto-generated patches, is a critical issue in APR. An overfitting patch does not fix the bug and can introduce new bugs. The work of Qi et al. [105] emphasises this fact regarding heuristic APR techniques. One reason for repair overfitting is the nature of the repair test suite. When the failing tests cannot define the bug well and/or the passing tests cannot define all of the correct behaviours, the patch does not properly fix the bug [113]. For example, if the repair test suite contains one failing test exercising the bug, the patch simply deletes the functionality exercised by this test [105].

Several studies have been carried out to mitigate repair overfitting in APR. *Patch prioritization* [114], i.e., sorting candidate patches based on the probability of correctness, is one direction of these studies. This technique can be applied when there is a lack of test

oracles or additional test cases to evaluate patch correctness. Most of patch prioritization approaches either consider the *semantic similarity* or *syntactic similarity* between the buggy program and the fixing element [114]. ObjSim [115] and CapGEN [116] are examples of semantic-similarity-based patch prioritization approaches. The works of Koyuncu et al.(IFixR) [117] and Jiang et al.(SimFix) [118] belong to syntactic similarity based approaches. In contrast to these two categories of works, the work of Kang et al. [119] considers the *naturalness* [120] of the patch. *ObjSim* [115] is a lightweight approach that uses only the program state at the exit point(s) to prioritize patches. The other approaches use external sources, such as bug reports (e.g. IFixR [117]), additional code repositories (e.g. Kang et al. [119]) and previously generated fixes (e.g. CapGen [116]).

Related to *test-driven* APR, another direction to mitigate repair overfitting is improving repair test suites. The works of Yu et al. [121], Yang et al. [113] and Xiong et al. [122] are examples of such approaches. All these works require an initial repair test suite. The repair test suite is systematically augmented in a way improving the quality of the patch. *UnsatGuided* by Yu et al. focuses on constrained-based repair techniques [121], while Yang's method [113] focuses on heuristic repair. Xiong's method [122] can be applied to both categories of repair techniques. However, this method needs a patch as an input in addition to a repair test suites.

If there is an automatic test oracle for a buggy program, more passing and failing test cases can be generated without human intervention, through which the quality of repair test suites can be improved to produce better repairs. Therefore, the findings of this thesis are helpful for producing better repairs for semantic bugs. In addition, the automatic oracles learned by our learning techniques contain some constraints related to program failures. These constraints could be useful for synthesising repair constraints in *constraint-based repair* techniques. Also, these could be applied to different tasks in *learning-based repair*. In addition, the automatic oracles could be improved to develop more precise correction criteria than repair test suites [9].

2.6 Summary

The key objective of this thesis is to develop learning techniques to generate automatic test oracles for semantic bugs. Such automatic test oracles could automate the process of detecting and fixing semantic bugs. Our literature review suggests that fixing semantic bugs is essential to avoid many of the adverse outcomes of software systems. Therefore, the findings of this thesis have significant importance in automated software testing.

According to this literature review, *test oracle automation* is still a developing research area and requires substantial improvement. Among the existing test oracle automation techniques, specified test oracles, i.e., those based on formal specifications, are ideal for semantic bugs. However, developing this kind of test oracle is difficult due to the difficulty of finding the formal specification of a software application practically. Therefore, derived test oracles, i.e., those derived from sources other than formal specifications, are most suitable for semantic bugs. In addition, some surveys related to test oracle automation provide a concrete pathway for the development of test oracles, which is useful for the present study.

There are a few works that use machine learning to develop automatic test oracles. These present some important insights for formulating our learning techniques. However, some limitations of these works make them difficult to apply to semantic bugs. For instance, existing machine learning-based oracle learning approaches pay no attention to obtaining training data systematically from a human. When the only source to obtain training data is a human, there are some significant issues to be considered. This study mainly focuses on these issues in developing oracle learning techniques.

Our oracle learning techniques could improve both *automated debugging* and *automated program repair* on semantic bugs. These improvements will be helpful in significantly reducing the manual effort required to find and fix semantic bugs in software systems.

Chapter 3

Research Methodology

3.1 Developing Oracle Learning Techniques

The objective of our oracle learning techniques is to *derive test oracles* [2] for semantic bugs by facilitating a systematic interaction with the human (the user or developer). In this learning setup, the human's task is to decide whether a test is *passing* or *failing* given the input and the program output. Even a person without programming knowledge can participate in this learning process if he knows the expected behaviour of the program.

As described in Section 2.3 of Chapter 2, Pezzè et al. [23] suggest the following four (4) steps for constructing a test oracle.

i. Identify the source of the information needed to derive the oracle.

ii. Recognize the program behaviour to be checked.

iii. Translate the source of information and the program behaviour into forms that can be checked against each other.

iv. Execute the oracle.

Our semi-automatic learning techniques follow these steps in generating automatic test oracles. The source of information for our automatic oracles is the human (the user or developer), as only the human can answer about a test failure in a semantic bug. Our learning techniques only use test cases labelled by the human as *passing* and *failing* to

derive oracles. In the beginning, we assume there is only one failing test. To obtain more labelled test cases, the learning techniques employ test generation techniques and systematically interact with the human. In our setup, the behaviour of the program's output under the test inputs is considered. Our learning techniques asks the human: *"For the input $\vec{i}$, the program produces the output o; is the bug observed?"*. If the answer is "Yes", the test case is a *passing* one; otherwise, it is a *failing* one. A user or a developer who knows the expected behaviour of the SUT can easily answer this kind of question. Our learning approaches use *supervised machine learning* [14] techniques to train a model working as an automatic oracle from the labelled test cases. In this manner, the human-labelled test cases are translated into a checkable/executable form. Given an unlabelled test case, the automatic oracle predicts whether it is *passing* or *failing*.

As described in Section 1.2, we assume that a failing input (f) of the semantic bug has been given, and there is a human oracle (with the knowledge of expected program behaviour) to check whether a test case is *passing* or *failing*.

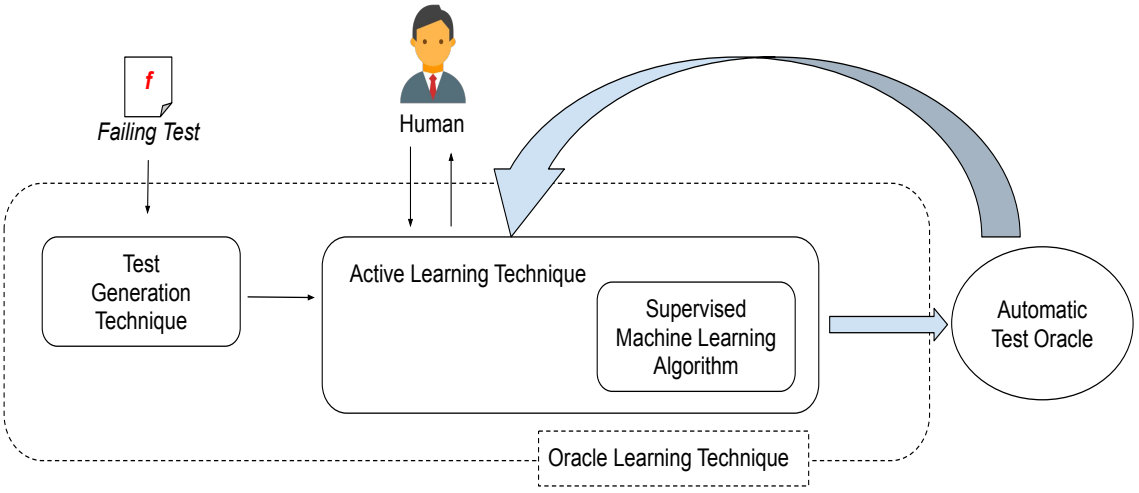


FIGURE 3.1: Architecture of an oracle learning technique. An oracle learning technique is a semi-automatic approach. At the end of the learning process, it returns an automatic test oracle

In developing an oracle learning technique, we follow the architecture shown in Figure 3.1. The main components of a learning technique and their roles are as follows.

1. *Test generation technique:* As one failing test is not enough to learn an oracle, we use a test generation technique to generate an adequate amount of training

data (test cases). McMinn et al. [59] suggest that the simplest way to incorporate human knowledge into test generation is to use a human-labelled test input as the seed in the beginning. Following this concept, the test generation process begins from a given failing test (f).

2. *Active learning technique:* We use an active learning technique [123] to systematically interact with the human and learn an oracle with less training data than would otherwise be needed. The active learning technique avoids the human receiving an unnecessary amount of labelling queries. *Membership query synthesis* methods and *stream-based sampling* methods are the two categories of active learning that we focus on in developing an oracle learning technique. *Pool-based sampling* methods are not considered, as we assume that only one failing test has been given. When *membership query synthesis* is used, the active learning itself generates test cases and directly presents them to the human. When *stream-based sampling* is used, the *test generation technique* generates test cases one by one, and the active learning technique selects test cases that are advantageous in oracle learning to present the human. In both of these approaches, the status of the automatic oracle being trained is considered.

3. *Supervised machine learning algorithm:* We train the automatic oracle as a classifier. The *supervised machine learning algorithm* trains the classifier using the human-labelled tests. The *active learning technique* decides whether to send a test case to the human based on the characteristics of the machine learning model. We select a machine learning algorithm based on the type of input that we address in the oracle learning technique.

The learning process begins with a single failing test (f) of the semantic bug. The *test generation technique* generates more tests, taking f as the starting point. The *active learning technique* maintains the systematic interaction between the human and the oracle learning process. The *supervised machine learning algorithm* trains a classifier using the human-labelled tests. The oracle learning technique incrementally trains the automatic oracle as human-labelled tests are received. The *active learning technique* considers the current status of the automatic oracle in sending the next labelling query to the human. At the end of the process, we obtain an automatic test oracle for the

semantic bug. This automatic oracle approximates the failure condition of the bug, hence a *bug oracle*

3.2 Machine Learning

Machine learning has been applied to automate various tasks in software engineering. It is also an important component of our learning technique architecture (Figure 3.1). In our setup, we use training data labelled (annotated) as *passing* or *failing*. The automatic oracles should predict whether a test case is passing or failing. Hence, *supervised machine learning* [124] techniques are suitable for training such automatic oracles from the labelled training data.

3.2.1 Classification Algorithms

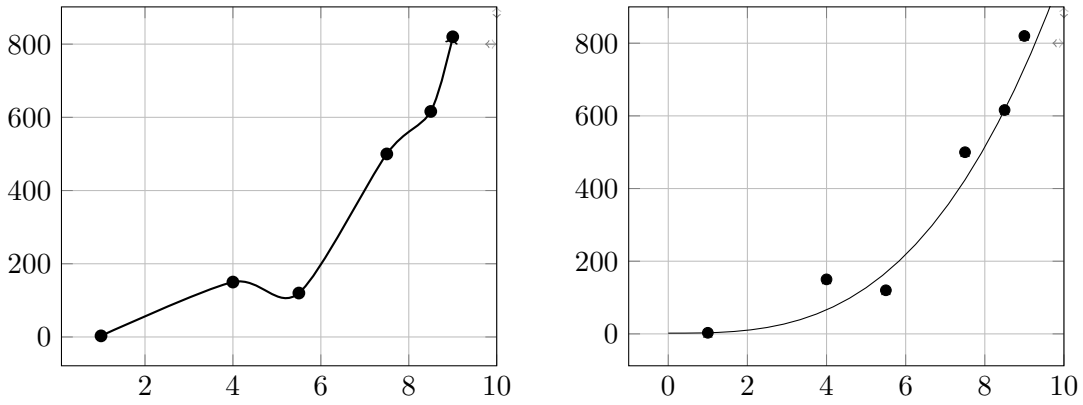
Classification algorithms are a type of supervised machine learning [14] and are able to derive classifiers from numeric data. A classification algorithm learns a *mapping function* as a classifier from a training dataset. The mapping function indicates the relationship between the features and class labels [125]. In *multi-class classifiers*, there are more than two target classes. A classifier used to classify news articles in terms of sports, politics and science is an example multi-class classifier. *Binary classifiers* are trained to predict only two classes. A spam filter distinguishing legitimate and spams emails belongs to this category. For a semantic bug associated with numeric inputs, an automatic oracle can be trained as a binary classifier.

Classification algorithms use various methods to learn classifiers, and the classifiers are represented in different forms. Based on their representation, classification algorithms can be divided into two types: *interpolation-based* [126] and *approximation-based* [127].

Given a set of training data, *Interpolation-based* approaches learn a model that exactly fits all the data points (e.g. Figure 3.2(a)). Furthermore, if the training dataset is T and f is the trained model, then $f(x)$, where $x \in T$, gives the correct label of x. *decision tree* algorithm [128] is an example of this category of approach. This algorithm can learn a set of constraints from a given dataset in terms of its features. The learned constraints take the form of inequalities, which are represented as a tree structure. This capability

of *decision tree* algorithm has been used in software testing tasks such as *fault localization* [66]. In addition, the *Incremental SMT Constraint Learner(INCAL)* [129] belongs to this category. Given a set of positive and negative examples, it learns a *Satisfiability Modulo Theory(SMT)-Linear Arithmetic* [130] formula, which is a set of constraints, satisfied only by the positive examples. Thus, *INCAL* can be used to develop SMT formula-based binary classifiers. SMT formulas are used in many areas in automated software testing, such as *symbolic execution* [62].

Approximation-based approaches estimate a model that fits a subset of training data in the best possible way, minimizing the error with the other data points (e.g. Figure 3.2(b)). *Support vector machines (SVMs)* [131] and *artificial neural networks(ANNs)* [132] are some popular approaches in this category. *SVM* has been more successful in many binary classification problems than other supervised learning approaches. Artificial neural networks have been applied in diverse domains for approximation tasks. However, in applying an ANN, there are many design decisions to be considered regarding the problem to be solved (e.g., the number of layers, number of neurons per layer, representation of the data, etc.).



(a) Interpolation (b) Approximation

FIGURE 3.2: Interpolation vs approximation for the same set of points

A set of classifiers can be collectively used to achieve better prediction accuracy, which is called *ensemble* learning [133]. *Bagging*, *boosting* and *adaptive boosting (AdaBoost)* are some popular ensemble algorithms [134]. Both types of classification algorithms (interpolation and approximation) can be used with ensemble learning. *Random forest* [135] is an example in which *bagging* is applied to a set of decision trees. The work of Li et al. [136] is an example that SVM is used with *AdaBoost*. This approach creates a

series of SVMs such that the errors of one classifier are resolved in subsequent classifiers. Finally, the series of classifiers is linearly combined into a single classifier.

Classification algorithms are helpful in addressing **RQ.1**, as they can learn classifiers from numeric data (unstructured inputs). Some of these techniques have been applied to develop automatic test oracles (Section 2.4). This study proposes a new architecture for oracle learning (Figure 3.1). Thus, the performance of the new oracle learning architecture under different classification algorithms should be explored.

3.2.2 Grammar Inference Algorithms

Grammar inference is an area of machine learning that learns patterns among strings as *formal grammars* [137]. Grammar inference algorithms have been applied in software engineering to tasks such as domain-specific language development [138] and log file analysis. There are *unsupervised* and *supervised* grammar inference approaches [124, 139].

For a semantic bug associated with string inputs, a grammar can be used as an automatic test oracle. For example, a grammar describing the pattern of the failing inputs can work as a test oracle for a bug. As we use training data labelled as passing and failing, supervised grammar inference algorithms are suitable to derive this kind of grammar.

Regular positive and negative inference (RPNI) [140] and *GOLD* [141] are conventional regular grammar [142] inference techniques. Given a set of positive and negative examples, these algorithms infer a *deterministic finite automaton (DFA)* [137] that accepts all the positive examples while rejecting the negative ones. The DFA represents a regular grammar. Higuera et al. [140] show that GOLD algorithm has an issue in the step of "filling holes".

Few works have been conducted on *context-free grammar* (CFG) [137] inference based on examples. Inductive CYK [143] and Imada et al. [144] are examples of context-free grammar inference approaches. Similar to RPNI and GOLD, these algorithms also use a set of positive and negative examples. Inductive CYK [143] incrementally develops production rules. If any production rule leads to the acceptance of any negative example, the algorithm backtracks to the previous point and considers another direction. The work of Imada et al. [144] solves a Boolean satisfiability problem to learn context-free

grammar from positive and negative examples. In contrast to these works, Segovia et al. [145] and Siegfried et al. [146] propose methods for inferring context-free grammars based only on a set of positive examples. The study of Moses et al. [147] discusses the applicability of using compression algorithms for context-free grammar inference. This problem is known as the *smallest grammar problem*. All the algorithms discussed in this work use only one string for context-free grammar inference, thus generating overfitting grammar. In addition to these, the works of Kim et al. [148] and Tu et al. [149] focus on inferring *probabilistic context-free grammar* (stochastic context-free grammar) [150].

Many recent grammar inference techniques use *genetic programming* [151]. The studies [152–157] are examples of such grammar inference techniques. Thomas et al. [152] and Bill et al. [155] focus on inferring probabilistic context-free grammar. Similar to Moses' work [147], Li et al. [154] address the smallest grammar problem. Unlike other genetic programming-based approaches, Rodrigues et al. [153] use new genetic operators in the grammar inference process.

The alphabet of the target language, which should be pre-determined, is an important consideration in most of these algorithms. As the alphabet gets larger, the number of examples required for accurate grammar inference significantly increases. This is a critical issue when there is a limited capability to obtain training data.

Formal grammars [137] are capable of representing the patterns of strings. They can also be extended to represent the patterns in many structured inputs. Thus, *grammar inference* techniques are helpful in addressing **RQ.2**. However, it is necessary to explore ways of adapting grammar inference algorithms to our oracle learning architecture.

3.2.3 Active Learning

Active learning is a branch of machine learning that focuses on achieving higher accuracy by selecting the most informative training data [123, 158]. It allows the learning of classifiers with less training data. There are three types of active learning scenarios: membership query synthesis, stream-based sampling and pool-based sampling.

In *membership query synthesis*, the active learning technique generates instances from the region of uncertainty of the classifier [159] and sends them to the oracle. The training

process continues with those labelled instances. L^* [160], ID [161] and IDS [162] are some active grammar inference algorithms that use membership queries.

Both *stream-based sampling* and *pool-based sampling* assume that obtaining unlabelled instances is inexpensive. In *stream-based sampling*, unlabelled instances are generated one at a time, and the learning technique decides whether to label the instance. The works of Zhu et al. [163], Chu et al. [164] and Zhang et al. [165] are examples of approaches to guiding instance selection from a data stream. *Pool-based sampling* methods assume that there is a small set of labelled instances and a large pool of unlabelled instances. The pool of unlabelled instances is considered static (non-changing). The learning technique selects instances for labelling based on an informativeness measure that evaluates all of the unlabelled instances in the pool. The works of Holub et al. [166], Joshi et al. [167] and Li et al. [168] are examples of pool-based sampling approaches.

As described in Section 3.1, *active learning techniques* can maintain a systematic interaction with the human. Thus, they are helpful in addressing both **RQ.1** and **RQ.2**. Also, the way that the active learning technique selects training data affects the quality of the repair test suites expected to be developed in relation to **RQ.3**.

3.3 Experimental Setup

We conduct experiments to evaluate the solutions explored to answer the research questions (Section 1.2.1). Programs with semantic bugs are used as the subjects of the experiments. We select such programs from program-repair benchmarks. (Appendix-Appendix B shows some details about the benchmarks used in the experiments).

To evaluate the oracle learning techniques developed to answer **RQ.1** and **RQ.2**, we select the benchmarks satisfying the following criteria.

1. There should be programs with real-world defects leading to semantic bugs.
2. For each faulty program, there should be a version where the bug has been fixed, i.e., *golden version*. The golden version demonstrates the expected, correct program behaviour.

3. For each faulty program, there should be a manually created, labelled test suite. This labelled test suite should contain at least one *failing test case*, i.e, a test case exposing the fault.

Depending on the nature of the learning technique, we consider more criteria in selecting the subjects (programs) for the experiments from a benchmark. For example, if the learning technique works with numeric inputs, we select the programs taking numeric inputs from a benchmark as the experimental subjects.

We use the *golden version* of a faulty program to simulate the human in the experiments. Given an input, if the faulty program and its golden version produce different outputs, we consider the test as *failing*; otherwise, it is *passing*(Figure 3.3). This method has been influenced by *differential testing* [169]. Differential testing uses different independent implementations (two or more) of the SUT. Given an input, the results of the implementations are compared to determine whether the test is passing or failing. The setup in Figure 3.3 is similar to this process.

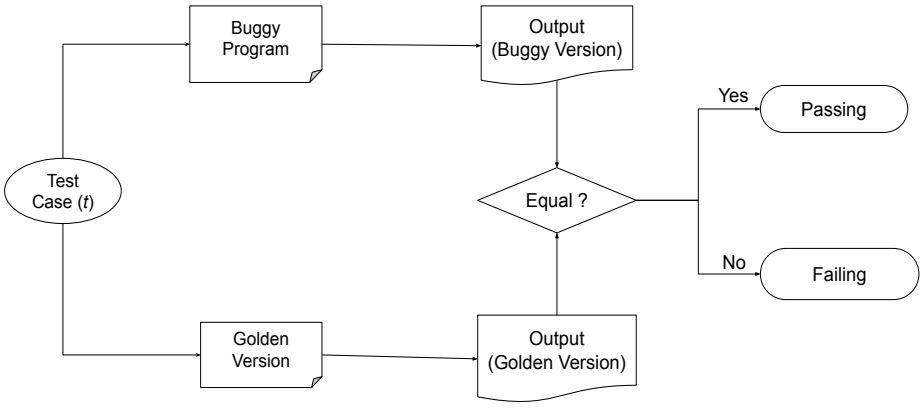


FIGURE 3.3: Usage of a buggy program and its golden version to simulate the *human oracle*

In the benchmark selection criteria, we consider that there should be a labelled test suite containing at least one failing test case for each faulty program. In applying an oracle learning technique to a faulty program, an input failing test (see Figure 3.1) is randomly selected from the labelled test suite. After the automatic oracle is generated, we allow it to predict the labels of all the test cases in the same test suite. No overfitting occurs

in this procedure, as our learning techniques generate training data by themselves (*Test generation technique* - Figure 3.1).

Related to **RQ.3**, we conduct some experiments with automated program repair techniques. In selecting benchmarks for these experiments, we consider the following additional criterion.

* For each faulty program, there should be a *repair validation* test suite.

In the program repair experiments related to **RQ.3**, we use the repair validation test suite to evaluate the patches.

As our learning techniques and the experiments involve many random operations, we repeat each experiment 30 times for each subject (faulty program).

3.4 Evaluation Metrics

To assess the effectiveness of the oracle learning techniques developed to answer **RQ.1** and **RQ.2**, we concentrate on the quality of the automatic oracles and the human effort associated with the learning process. The following evaluation metrics are used to evaluate the oracle quality and human effort.

1. *Accuracy*: Percentage of correctly predicted test cases by the automated oracle (Equation 3.1)
2. *Conditional accuracy - failing / Recall for failing test cases* : Percentage of correctly predicted failing test cases from the actual failing test cases. (Equation 3.2)
3. *Conditional accuracy - passing/ Recall for passing test cases* : Percentage of correctly predicted passing test cases from the actual passing test cases. (Equation 3.3)
4. *Precision - failing* : Percentage of correctly predicted failing test cases from the test cases predicted as failing. (Equation 3.4)
5. *Precision - passing* : Percentage of correctly predicted passing test cases from the test cases predicted as passing. (Equation 3.5)

6. *Human labelling effort* : Number of test cases labelled by the human in the training process.

$$Accuracy = \frac{Number\ of\ correctly\ predicted\ test\ cases}{Number\ of\ test\ cases\ in\ the\ test\ suite} \tag{3.1}$$

$$\begin{matrix} Conditional\ Accuracy \\ -Failing \end{matrix} = Recall\text{-}Failing = \frac{\begin{matrix} Number\ of\ correctly\ predicted \\ failing\ test\ cases \end{matrix}}{\begin{matrix} Number\ of\ failing\ test\ cases \\ in\ the\ test\ suite \end{matrix}} \tag{3.2}$$

$$\begin{matrix} Conditional\ Accuracy \\ -Passing \end{matrix} = Recall\text{-}Passing = \frac{\begin{matrix} Number\ of\ correctly\ predicted \\ passing\ test\ cases \end{matrix}}{\begin{matrix} Number\ of\ passing\ test\ cases \\ in\ the\ test\ suite \end{matrix}} \tag{3.3}$$

$$Precision\text{-}Failing = \frac{\begin{matrix} Number\ of\ correctly\ predicted \\ failing\ test\ cases \end{matrix}}{\begin{matrix} Total\ number\ of\ test\ cases \\ predicted\ as\ failing \end{matrix}} \tag{3.4}$$

$$Precision\text{-}Passing = \frac{\begin{matrix} Number\ of\ correctly\ predicted \\ passing\ test\ cases \end{matrix}}{\begin{matrix} Total\ number\ of\ test\ cases \\ predicted\ as\ passing \end{matrix}} \tag{3.5}$$

The first five metrics are related to the oracle quality. These are computed by comparing the actual labels of the test cases with those predicted by the automatic oracle. We especially consider *conditional accuracy-failing* (recall-failing) in addition to *accuracy*, as the identification of failing test cases is essential for a test oracle to detect bugs. We incorporate some methods into our learning techniques to deal with the *class imbalance problem*. These five metrics collectively indicate the success of these methods.

Our oracle learning techniques use certain techniques to systematically involve the human in training, through which we expect to reduce the interaction with the human as much as possible. We measure the effectiveness of these techniques in terms of *human labelling effort*.

We use the following metrics in our automated program repair experiments that are conducted related to **RQ.3**.

1. *Repairability*: Percentage of the subjects repaired by the automated program repair (APR) technique.
2. *Validation Score*: If the APR technique produces a patch, the proportion of repair validation tests that pass on the patched program.

Sometimes, the APR technique does not produce a repair with the given repair test suite. Hence, we concentrate on *repairability*. If the APR technique generates a patch, we measure the validation score to assess the quality of the patch.

We introduce additional metrics in the experiments to evaluate different properties of the learning techniques. Also, we use *Wilcoxon Test* [170] to compare some results based on the statistical significance. Wilcoxon test is a non-parametric statistical test, and it is suitable for our experimental environments.

Chapter 4

Learning Automatic Test Oracles for Unstructured Inputs

This chapter explores an answer to **RQ.1**: developing an oracle learning technique for semantic bugs in programs taking *unstructured inputs*. *Numeric inputs* instantiate unstructured inputs. Therefore, we present an approach called LEARN2FIX to learn automatic test oracles for the semantic bugs in programs taking inputs. This approach needs only one failing test case of the bug and systematically interacts with the human (the user or developer) to obtain the training data necessary to learn an oracle. In oracle learning, LEARN2FIX derives the condition under which the semantic bug is exposed; i.e., the *failure condition* of the semantic bug. Given the input and the corresponding output of the buggy program, an automatic oracle generated by LEARN2FIX predicts whether a test case is passing or failing.

4.1 Unstructured Inputs

Numeric inputs are the most widely used unstructured inputs in programs. The validity of a numeric input is always determined with respect to a domain. As an example, consider the number 3.56. This is a *valid* number in the domain of *real numbers* ($\mathbb{R}$) [171]. The same number is *invalid* in the domain of *natural numbers* ($\mathbb{N}$). The domain of a set of numbers can be constrained by mathematical and logical operators. For example,

the domain of natural numbers can be constrained to have natural numbers from 1-100 as follows.

- $S = \{x : x \in \mathbb{N} \land x \leq 100\}$

This concept is useful in designing an automatic oracle for a semantic bug in a program taking numeric inputs. The numeric passing and failing test cases of a semantic bug can be separated by defining constraints like this.

4.2 Motivation

We use the buggy program in Listing 4.1 to demonstrate the challenge of learning automatic oracles for programs that take numeric inputs. This example is taken from an experiment conducted by Russ Williams [172]. In this experiments, 12 participants were asked to write programs for solving the *Triangle Classification Problem*; i.e., classifying triangles as *equilateral*, *isosceles*, *scalene* or *invalid* given the lengths of their sides.

```
1  int f_steve_classify(int a,int b,int c){
2          if(a<=0 || b<=0 || c<=0)
3                  return 4;    //Invalid
4          if(a<=c-b || b<=a-c || c<=b-a)
5                  return 4;    //Invalid
6          if(a==b==c)      //BUG !
7                  return 1;    //Equilateral
8          if(a==b || b==c || c==a)
9                  return 2;    //Isosceles
10         return 3;
11         }
```

LISTING 4.1: A buggy C program for triangle classification

`f_steve_classify` function takes 3 inputs that represent the lengths of the sides of a triangle and returns an integer where the return value

- 1 means it is equilateral (all sides equal in length)
- 2 means it is isosceles (exactly 2 equal sides)

- 3 means it is scalene (no equal sides)
- 4 means it is an invalid triangle

The C program in Listing 4.1 is Steve's implementation of triangle classification, which has a bug in Line 6. The programmer uses the C statement `a==b==c` (Line 6) instead of `a==b && b==c` to check whether the triangle is equilateral. Thus, given the input $t = \langle 2, 2, 2 \rangle$, Line 6 evaluates it as follows.

`(2==2==2)` $\rightarrow$ `((2==2)==2)` $\rightarrow$ `((1)==2)` $\rightarrow$ `0`

The reason is that C represents the Boolean values `True` as 1 and `False` as 0, and thus `2==2` $\rightarrow$ `1` and `1==2` $\rightarrow$ `0`. Therefore, Listing 4.1 is incorrect for:

i. All *equilateral triangles*, except $\langle 1, 1, 1 \rangle$

ii. All *isosceles triangles* where `c=1`

For test input t, Listing 4.1 returns 2 (isosceles), while we expect it to return 1 (equilateral). Also, the program does not crash. This is a *semantic bug*, as the program shows a deviation from the expectation for test input t. Due to the difference between the actual and expected output, we identify t as a *failing test case*.

The program failure exposed by t can be identified only if we know the expected, correct output (*1:equilateral*) that Listing 4.1 should produce for t. Similarly, it is essential to know the expected, correct program behaviour of the *program under test (PUT)* to detect semantic bugs. For this reason, only a human (the developer or the user) can detect this category of bugs.

Usually, a semantic bug is reported with a single failing input triggering the bug. Assume that the user has found the failing input $t = \langle 2, 2, 2 \rangle$. Indeed, only this failing input is insufficient to identify why the program in Listing 4.1 fails to produce the correct outputs for certain inputs. Similarly, this bug cannot be detected and fixed automatically with a single failing input. To locate the failure, both automated program repair and automated debugging techniques need more passing and failing inputs. In APR, more labelled (as passing and failing) test inputs are required to validate the generated fixes. However, finding more passing and failing test cases with human intervention is impractical.

The solution to all the challenges described above is to develop automatic test oracles. However, we have only one failing input and the human to differentiate the test cases as *passing* or *failing*. The method of learning an automatic oracle should be able to cope with such circumstances.

4.3 Background

Our objective is to develop an automatic test oracle for a given semantic bug, beginning from one *failing* test case. We develop the automatic test oracle as a classifier that categorizes test cases as *passing* and *failing*, applying classification algorithms in machine learning. To train such a classifier, we need a set of *passing* and *failing* test cases as a training dataset. In this scenario, the human (the user or the developer) is the only way to query the label of a test case.

Indeed, a classification algorithm cannot train an accurate classifier as a test oracle using one failing test case. Thus, more test cases should be generated as training data. Also, one failing input is insufficient to explore the condition under which the bug is exposed. For this reason, the test generation should focus on producing sufficient failing tests to support oracle learning.

There are many classification algorithms in machine learning to develop classifiers with numeric data. Support vector machines (SVM), decision trees, naïve Bayes, and artificial neural networks are examples of popular algorithms that can develop classifiers for numeric data [14]. Several classification algorithms have been applied to develop test oracles from numeric data (e.g. artificial neural networks in Jin et al. [15] and Shahamiri et al. [71]). These oracle learning approaches are passive learning approaches (Section 2.4). In contrast, we follow an active oracle learning architecture as in Section 3.1. Therefore, we need to analyse the applicability of classification algorithms to our active oracle learning architecture.

Some classification algorithms might need a large amount of training data in test oracle learning. In this problem, it is necessary to generate and label test cases to obtain training data. As the human labels test cases, obtaining a large training dataset is impractical. Thus, learning an accurate classifier with less training data is a significant

concern when training test oracles for semantic bugs. We either have to select classification algorithms satisfying this condition or introduce additional techniques to learn more accurate test oracles with less training data.

Having a balanced training dataset, i.e., equal amounts of data from each class, is essential for most classification algorithms to develop accurate classifiers. Otherwise, the training process is susceptible to the *class imbalance problem* [17]. In many semantic bugs in programs taking numeric inputs, the inputs exposing the bug are rarely observed. If numeric inputs were randomly generated, the majority of the inputs would be passing. Due to this situation, obtaining a training dataset with equal amounts of passing and failing test cases is difficult. Consequently, the *class imbalance problem* affects oracle learning by classification algorithms.

Mutational fuzzing is an effective test generation method that can generate test cases in the neighbourhood of a failing test case. Exploring test cases in the neighbourhood of a failing test case helps to collect more details about the location and behaviour of the bug. The success of this approach has been proven in the coverage-based, mutational greybox fuzzer American Fuzzy Lop (AFL) [173], which generates more crashing inputs by mutating a seed crashing input. Following the same concept, mutational fuzzing can be applied to generate more failing test cases by exploring the neighbourhood of a failing test case. The neighbourhood test cases given by mutations reveal the boundaries of the bug, i.e., how the program's behaviour changes from buggy to correct and vice versa, under small changes to the input. All these abilities of mutational fuzzing are useful to address the *Class Imbalance Problem*.

Active learning [123] techniques are able to learn classifiers with less training data and are applicable when obtaining labelled data is expensive (Section 3.2.3). Thus, we can use these techniques to learn test oracles for semantic bugs. The method of Holub et al. [166] is an active learning technique that selects data points to be labelled based on the current status of the classifier being trained. This method selects the *most informative unlabelled data point*(MIUP) from the unlabelled data set. Holub's method can facilitate systematic human involvement in oracle learning. Also, it can be modified to cope with the *class imbalance problem*.

4.3.1 Preliminary Analysis on Training Oracles With Some Classification Algorithms

We analysed the capability of *SVM*, naïve Bayes, and decision trees [14] classification algorithms to train a test oracle with little training data. The programs with semantic bugs were selected from the benchmark *Triangle Classification* [172]. This benchmark has different test suites. For each program, we randomly selected 30% of the test cases, including at least one failing test case, from each test suite as the training dataset. The rest of the test cases (70%) were used as the validation dataset. The training and validation datasets contained *more passing test cases than failing ones*. To evaluate the results, we measured the prediction accuracy and conditional accuracy-failing (recall-failing tests); i.e., the accuracy of predicting failing test cases.

The experimental results (Appendix A) suggest that the *SVM* and *decision trees* algorithms can produce classifiers with significant prediction accuracy even under a little amount training data. However, these algorithms do not train the classifiers to accurately identify failing test cases. Their lower conditional accuracy indicates this fact. Sometimes, the classifier is trained to predict everything as passing. As the validation datasets contain fewer failing test cases than passing test cases, we observe higher accuracy even though the failing test cases are misclassified.

The key reason for this issue is the *class imbalance problem* [17]. Furthermore, the classifiers are trained to predict passing test cases more accurately, as there are more passing test cases in the training datasets. Due to the inadequacy of the failing test case in the training datasets, the classifiers are not trained to predict failing test cases correctly. However, predicting test failures is an essential capability of a test oracle in automated debugging and repair.

This small experiment suggests the following.

i. Random selection of training data does not lead to high-quality classifiers (automatic oracles).

ii. The *class imbalance problem*, i.e., where there are very few failing tests and more passing tests, affects the ability to identify failing tests.

These issues should be addressed in designing an oracle learning technique for programs taking numeric inputs.

4.4 Methodology

With the buggy program ($\mathcal{P}$), our proposed approach for numeric inputs, LEARN2FIX, assumes that the following has been given.

1. One failing test case of the bug (f)
2. The human ($\mathcal{H}$) to answer whether a test is passing or failing

A test case t is of the form $t = \langle \vec{i}, o \rangle$, where $\vec{i}$ is a vector of input variable values and $o = \mathcal{P}(\vec{i})$ is the output of $\mathcal{P}$ for $\vec{i}$. Also, we assume that $\vec{i}$ has a fixed length, and the human can answer at most L queries.

Algorithm 1 shows an overview of LEARN2FIX. In designing this algorithm, we focused on addressing the issues identified in Section 4.3.1. The algorithm maintains two sets of test cases T for all human-labelled test cases and $T_{\text{✗}}$ for human-labelled failing test cases ($T_{\text{✗}} \subseteq T$). Firstly, using the given failing input (f), LEARN2FIX trains an automatic oracle ($\mathcal{O}$) by a classification algorithm. As it is trained with a single failing input, $\mathcal{O}$ at this point predicts everything as *failing*.

More training data are required to improve the accuracy of the automatic oracle ($\mathcal{O}$). Thus, LEARN2FIX randomly selects a failing test case (f') from $T_{\text{✗}}$ and applies arithmetic mutations it to generate a new test case (t) (Line 6 - Algorithm 1). If DECIDE2LABEL returns `true`, t is presented to the human oracle ($\mathcal{H}$) for labelling (Line 7 - Algorithm 1). Next, t is added to $\mathcal{T}$, and the automatic oracle ($\mathcal{O}$) is retrained with T (Line 13). if t is a *failing* test case, it is added to $\mathcal{T}_{\text{✗}}$ (Line 9). This process continues until the maximum number of labelling queries (L) is reached, or a timeout occurs.

4.4.1 Generating More Failing Test Cases

A set of human labelled passing and failing test cases is required to train a classifier as an automatic test oracle. LEARN2FIX uses *mutational fuzzing* [30] for this task. Because of

Algorithm 1 LEARN2FIX Active Oracle Learning

Input: Buggy program ($\mathcal{P}$), Failing test case ($f = \langle \vec{i}, o \rangle$)
Input: Human oracle ($\mathcal{H}$), Maximum labelling queries (L)
1: Failing test cases $T_{\text{✗}} \leftarrow \{f\}$
2: Labelled test cases $T \leftarrow \{f\}$
3: Automatic Oracle $\mathcal{O} \leftarrow$ TRAIN_CLASSIFIER(T)
4: **while** ($|T| < L$) and not timed out **do**
5: Failing test case $f^{'} \leftarrow$ RANDOM_SELECT($T_{\text{✗}}$)
6: Generate test case $t \leftarrow$ MUTATE_FUZZ($f^{'}$)
7: **if** DECIDE2LABEL($t, \mathcal{O}$) = `true` **then**
8: Human label $h = \mathcal{H}(t)$
9: **if** $h = fail$ **then**
10: Failing test cases $T_{\text{✗}} \leftarrow T_{\text{✗}} \cup \{t\}$
11: **end if**
12: Labelled test cases $T \leftarrow T \cup \{t\}$
13: Automatic Oracle $\mathcal{O} \leftarrow$ TRAIN_CLASSIFIER(T)
14: **end if**
15: **end while**

the numeric inputs, LEARN2FIX applies *arithmetic mutations* [174] to f to generate new test cases. This process explores the neighbourhood of f and has a higher probability of generating failing test cases compared to *generational fuzzing* [29].

For example, assume that for the motivating example in Listing 4.1, $\langle 2, 2, 2 \rangle$ is the given failing input $f = \langle \langle 2, 2, 2 \rangle, 2 \rangle$. Also, assume that for each position a in $\vec{i}$, we employ one of five arithmetic mutation operators chosen uniformly at random: $\vec{i}[a] = \vec{i}[a]$, $\vec{i}[a] = \vec{i}[a] + 1$, $\vec{i}[a] = \vec{i}[a] - 1$. The following test cases are generated when actually running this fuzzer on f.

$$\langle \langle 2, 2, 1 \rangle, 1 \rangle_? \quad \langle \langle 1, 3, 3 \rangle, 2 \rangle_?$$
$$\langle \langle 1, 3, 2 \rangle, 4 \rangle_? \quad \langle \langle 3, 3, 1 \rangle, 1 \rangle_?$$
$$\langle \langle 2, 1, 3 \rangle, 4 \rangle_? \quad \langle \langle 3, 3, 3 \rangle, 2 \rangle_?$$
$$\langle \langle 2, 1, 1 \rangle, 4 \rangle_? \quad \langle \langle 1, 2, 3 \rangle, 4 \rangle_?$$
$$\langle \langle 3, 2, 2 \rangle, 2 \rangle_? \quad \langle \langle 2, 3, 2 \rangle, 2 \rangle_?$$

Three out of ten (3/10) test cases generated above are *failing*, i.e., $\langle \langle 2, 2, 1 \rangle, 1 \rangle$, $\langle \langle 3, 3, 1 \rangle, 1 \rangle$, and $\langle \langle 3, 3, 3 \rangle, 2 \rangle$. In contrast, assume that we randomly generate three numbers in the range $[-2^{63}, 2^{63} - 1]$. The probability of finding a test case representing an isosceles triangle with $c = 1$ or an equilateral triangle with $c \neq 1$ is extremely low. Thus, *mutational fuzzing* has a higher probability of generating failing test cases than *generational*

fuzzing. Moreover, these test cases are in the vicinity of f and demonstrate how the program's behaviour changes from buggy to correct and vice versa under small changes. Studying the bug is easier with these test cases than randomly generated passing and failing test cases.

The above example demonstrates that *mutational fuzzing* can generate more failing inputs. Also, the generated test cases are helpful in studying the bug. These abilities of *mutational fuzzing* partially address the *class imbalance problem*.

4.4.2 Training a Classifier as a Test Oracle

To compute the automatic oracle, LEARN2FIX trains a binary classifier based on a human labelled training dataset. The function TRAIN_CLASSIFIER uses the same classification algorithm in both Algorithms 1 and 2. We consider the input ($\vec{i}$) and the output (o) values of a test case as the *features* used in the classification algorithm. We consider *passing* and *failing* as the two classes to be predicted. The function TRAIN_CLASSIFIER uses test cases labelled by the human (T) to train a binary classifier as the automatic oracle ($\mathcal{O}$). Given a test case, an automatic oracle ($\mathcal{O}$) predicts the label based on the input ($\vec{i}$) and corresponding buggy program output (o).

Usually, a classification algorithm requires at least one data point from each class. However, human-labelled test suites (training test suites) containing only failing tests can be generated in oracle learning. If so, we assume that TRAIN_CLASSIFIER returns a classifier that predicts every test case as *failing*.

There are many classification algorithms in machine learning that train binary classifiers from numeric data. As described in Section 3.2.1, based on the classifier's representation, these algorithms can be divided into two groups as *interpolation-based* and *approximation-based*. *Interpolation-based* approaches model that exactly fits the training data. In contrast, *approximation-based* methods estimate a model that fits a subset of training data, minimising the error with the other data points.

AdaBoost and *decision Tree* are examples of *interpolation-based* classification algorithms. The work of Braga et al. [72] uses *AdaBoost* algorithm to develop test oracles; however, their method is domain-specific. Also, the survey paper of Briand et al. [22] suggests that *decision trees* are effective in modelling the failure condition of a bug. Due to

these reasons, we evaluated the performance of *AdaBoost* and *decision tree* classification algorithms with LEARN2FIX. *AdaBoost* is an ensemble learning algorithm [133]. In the experiments, we used *decision trees* as the base estimator of *AdaBoost*. In addition, we selected the *Incremental SMT Constraint Learner* (INCAL) [129], which generates interpolation binary classifiers as Satisfiability Modulo Theory (SMT) [130] formula. *Symbolic Execution* [175] uses SMT constraints to group the inputs that exercise a particular path. Thus, SMT formula can be used to group the failing and passing inputs of a bug. For this reason, we selected INCAL [129] as a classification algorithm for our experiments.

Artificial neural networks and *support vector machines*(SVM) are examples of *approximation-based* classification algorithms. The work of Jin et al. [15] uses two *artificial neural network* setups to generate automatic test oracles. One setup has two hidden layer with 20 and 5 neurons (MLP(20,5)). The other setup has only one hidden layer with 20 neurons (MLP(20)). We selected these neural network configurations for our experiments. In addition, we chose SVM and *naïve bayes* under approximation-based classification algorithms. SVM is an algorithm that can be used in high-dimensional or infinite-dimensional space [176]. *naïve bayes* is based on the Bayes theorem. This classification algorithm is able to learn an accurate classifier with relatively little training data [176]. These two algorithms have been applied in different domains; however, their applicability to test oracle automation has not been explored.

The selected set of classification algorithms is as follows.

i. Support vector machine (SVM)

ii. Decision tree (DT)

iii. Naïve Bayes (NB)

iv. AdaBoost (ADB)

v. Incremental SMT constraint learner (INCAL)

vi. Artificial neural networks / Multi-layer perceptrons (MLP)

We experimentally evaluate the performance of these algorithms to know which category of classifier representation (interpolation or approximation) is most suitable for

LEARN2FIX. Moreover, we explore the best-performing classifier representation with LEARN2FIX.

4.4.3 Maximising the Probability of Labelling Failing Test Cases

Algorithm 2 DECIDE2LABEL

Input: Unlabelled test case $t_?$, Automatic Oracle $\mathcal{O}$
Input: Committee Size S
Let T be training test cases that $\mathcal{O}$ has been trained
Predicted label $\mathcal{L}_\mathcal{O} \leftarrow \mathcal{O}(t_?)$
if $\mathcal{L}_\mathcal{O} = \mathit{Failing}$ **then**
  return `true`
else
  $\mathit{fail\_votes} = 0$
  for $i \leftarrow 1$ to S **do**
    Generated test case $t'_? = \text{MUTATE\_FUZZ}(t_?)$
    $t'_{\checkmark} \leftarrow$ Assume that $t_?$ label as *Passing*
    $t'_{\times} \leftarrow$ Assume that $t_?$ label as *Failing*
    Hypothetical Oracle $\mathcal{O}_{\checkmark} \leftarrow$ TRAIN_CLASSIFIER$(T \cup \{t'_{\checkmark}\})$
    Hypothetical Oracle $\mathcal{O}_{\times} \leftarrow$ TRAIN_CLASSIFIER$(T \cup \{t'_{\times}\})$
    if $\mathcal{O}_{\checkmark}(t_?) = \mathit{Failing}$ **or** $\mathcal{O}_{\times}(t_?) = \mathit{Failing}$ **then**
      $\mathit{fail\_votes} \leftarrow \mathit{fail\_votes} + 1$
    end if
  end for
  $\widehat{\theta} = \frac{\mathit{fail\_votes}}{2 \times S}$
  if $\widehat{\theta} \geq 0.5$ **then**
    return `true`
  else
    return `false`
  end if
end if

As the minority class is *failing*, LEARN2FIX improves the classifier's ability to identify *failing* test cases, using the limited human queries. For this purpose, LEARN2FIX maximises the probability of labelling failing test cases in oracle learning. This strategy helps to address the *class imbalance problem*. To maximise the probability of labelling failing test cases, LEARN2FIX selects test cases with higher failure likelihood. Algorithm 2 (DECIDE2LABEL) describes this process. This method has been influenced by the work of Holub et al. [166]. Following Holub's method, the DECIDE2LABEL-algorithm estimates the failure likelihood based on the current status of the automatic oracle ($\mathcal{O}$).

The key concept in Holub's method is to select the *most informative unlabelled point*(MIUP) for labelling based on the current status of the classifier. Holub's method considers the

data point with the *minimum expected entropy* (MEE) [166] as the MIUP. In finding the data point with the MEE, Holub's method estimates the *look-ahead probability* of each class based on a *committee of classifiers* with hypothesized labels [166]. This is an active learning method based on *pool-based sampling* [123]; i.e., it assumes that there is a pool of unlabelled instances. In our setup, test cases are generated as a stream (one by one) by mutational fuzzing [30]. Therefore, in the DECIDE2LABEL-algorithm, we convert Holub's pool-based sampling approach to a *stream based* sampling method.

The DECIDE2LABEL-algorithm sends test cases predicted as *failing* by the automatic oracle being trained ($\mathcal{O}$) to the human for labelling. If the given test case ($t_?$) is actually *failing*, human labelling of $t_?$ allows $\mathcal{O}$ to learn more about the failure. If $t_?$ is actually a *passing* test case, it implies that $\mathcal{O}$ has not been trained correctly. In this case, LEARN2FIX rectifies $\mathcal{O}$ by human labelling of $t_?$ and using it in training.

If $\mathcal{O}$ predicts $t_?$ as *passing*, the DECIDE2LABEL-algorithm calculates the probability that $\mathcal{O}$ predicts $t_?$ as *failing*. Intuitively, there is an equal probability of classifying a test case into either class. LEARN2FIX estimates the probability that $\mathcal{O}$ predicts $t_?$ as *failing* one-step ahead. Following Holub's look-ahead probability estimation method [166], the DECIDE2LABEL-algorithm constructs a committee of automatic oracles (Line 7-16) for this task.

In creating the oracle committee, first, the DECIDE2LABEL-algorithm generates a new test case ($t'_?$) by applying mutational fuzzing to $t_?$. $t'_?$ is hypothetically labelled as *passing* ($t'_{\checkmark}$) (Line 9). Then, a new hypothetical oracle ($\mathcal{O}_{\checkmark}$) is trained with the training set $T \cup \{t'_{\checkmark}\}$ (Line 11, T: The initial training set of $\mathcal{O}$). The same test case is hypothetically labelled as *failing* ($t'_{\times}$)(Line 10). Another hypothetical oracle ($\mathcal{O}_{\times}$) is trained with the training set $T \cup \{t'_{\times}\}$ (Line 12). The DECIDE2LABEL-algorithm generates two (2) hypothetical oracles for a newly generated test case. Thus, in S fuzzing iterations, a committee containing $2 \times S$ automatic oracles is generated. Each hypothetical oracle created by adding a hypothetically labelled test case to the initial training (T) set demonstrates a possible status of the automatic oracle ($\mathcal{O}$) one step ahead. As each newly generated test case ($t_?$) is hypothetically labelled as both *passing* and *failing* and contributing to two different hypothetical oracles, the oracle committee is, overall, *unbiased*.

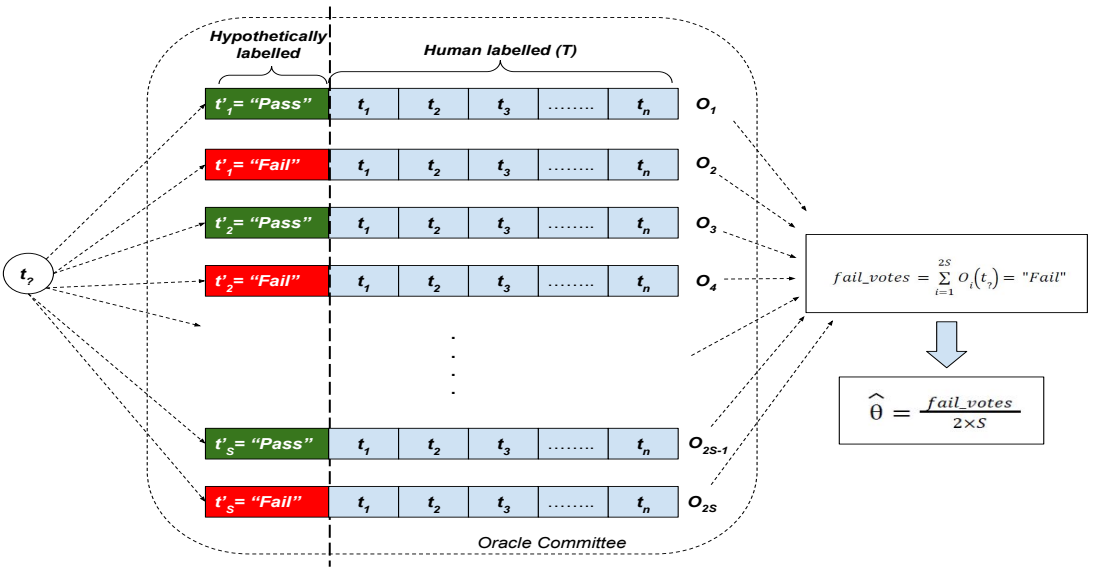


FIGURE 4.1: Calculating the failure probability via a committee of oracles

Finally, the unlabelled test case $t_?$ is presented to the oracle committee, and the number of occurrences that $t_?$ is predicted as *failing*, i.e., *fail_votes*, is counted. (Line 11-12). As there are $2 \times S$ oracles in the committee, the probability of labelling $t_?$ as *failing* is estimated by Equation 4.1 (Figure 4.1).

$$\widehat{\theta} = \frac{fail\_votes}{2 \times S} \tag{4.1}$$

As the oracles in the committee are some possible future states of $\mathcal{O}$, $\widehat{\theta}$ is a *look-ahead estimate* of the probability of *failing*. The DECIDE2LABEL-algorithm considers that test cases with $\widehat{\theta} \geq 0.5$ have a higher failure likelihood and sends them for human labelling (Line 18). According to the oracle committee, if $t_?$ has a higher failure probability, it implies that the automatic oracle ($\mathcal{O}$) has not been trained adequately to identify the failing test cases. Thus, labelling such test cases and using them in training rectify the automatic oracle ($\mathcal{O}$).

Algorithm 2 (DECIDE2LABEL) incrementally improves the automatic oracle that predicted everything as *failing* in the beginning. The automatic oracle ($\mathcal{O}$) gets improved as its capability to identify failing test cases (the minority class) is improved. As the oracle's accuracy for failing inputs increases, failing test cases are selected more frequently for human labelling. In this manner, the DECIDE2LABEL-algorithm addresses the *class imbalance problem*.

Figure 4.2 demonstrates the overall workflow of LEARN2FIX.

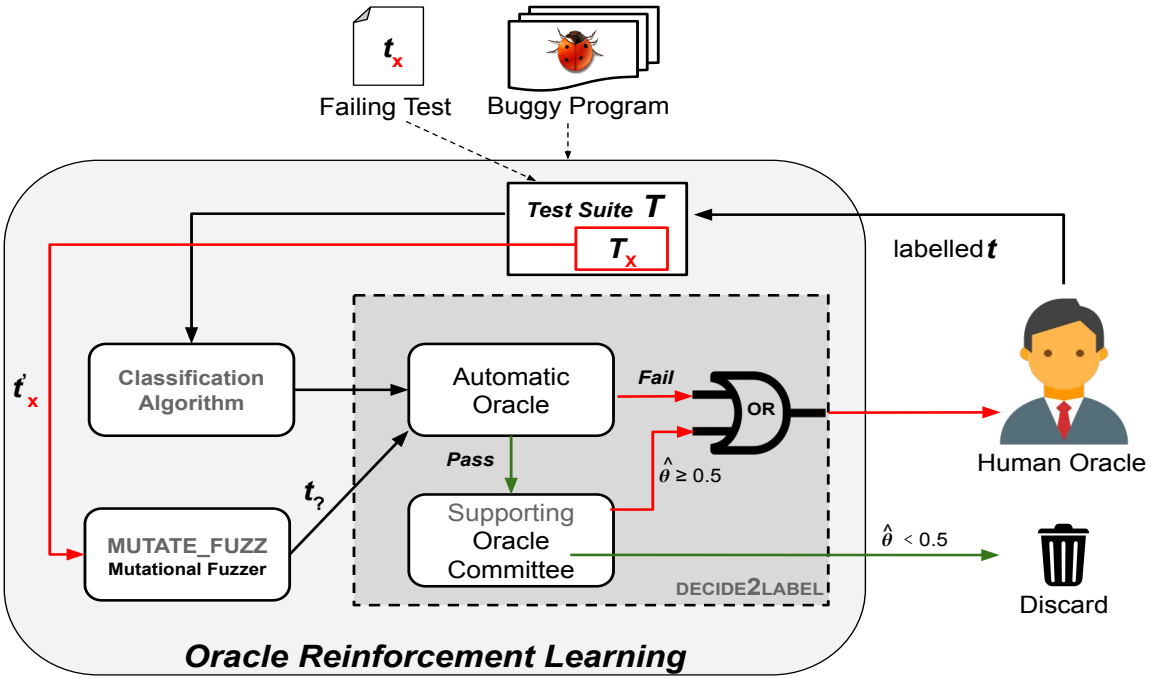


FIGURE 4.2: Workflow of LEARN2FIX

The components of LEARN2FIX can be mapped to the oracle learning in architecture in Figure 3.1 as in Table 4.1.

Test Generation Technique	Mutational Fuzzing - Section 4.4.1
Active Learning Technique	Stream-based sampling with a committee oracles - Section 4.4.3
Supervised Machine Learning Algorithm	Binary classification algorithms - Section 4.4.2

TABLE 4.1: Mapping of LEARN2FIX's components to the oracle learning architecture

4.5 Experimental Setup

We conducted several experiments to evaluate the performance of LEARN2FIX. In these experiments, we evaluated the following.

i. Oracle quality

ii. Human labelling effort

iii. The category of classifier representation most suitable for LEARN2FIX: *interpolation-based* or *approximation-based*,.

For oracle quality, we evaluated how accurately the automatic test oracles can predict the labels of test cases. Under human labelling effort, we assessed how LEARN2FIX uses the given limited number of queries in oracle learning. We selected a few classification algorithms from the *interpolation-based* and *approximation-based* categories (Section 4.4.2). Using those algorithms, we evaluated which category of classifier representation is most suitable for LEARN2FIX.

4.5.1 Experimental Subjects

To evaluate LEARN2FIX, we selected 552 programs from *Codeflaws* [177] benchmark according to the following criteria.

1. There should be a sufficiently large number of programs that are algorithmically complex.
2. There should be a diverse set of real-world defects that cause *functional bugs*; i.e., the programs produce incorrect or unexpected output for certain inputs. There should be one functional bug for each subject.
3. For each subject, there should be a *golden version*, i.e., a program that produces the expected, correct output for an input. For a given input we simulate the Human oracle ($\mathcal{H}$) by comparing the subject's (buggy program's) output with its golden version's output. If both the outputs are different, the human label of the test case is considered as *failing* (Figure 3.3).
4. For each subject, there should be a *manually constructed* and *labelled* test suite.
5. For each subject, there should be at least one failing test case in the human-labelled test suite, i.e., a test input for which the buggy program and its golden version produce different outputs. Otherwise, LEARN2FIX cannot be started.
6. For each subject, there should be test inputs having a constant number of numeric values. For each such test input, the program should produce a numeric output. Otherwise, the classification algorithms cannot be applied to learn the automatic oracle ($\mathcal{O}$).

We ignored IntroClass and ManyBugs benchmarks [178], as these do not satisfy our selection criteria. ManyBugs contains programs taking complex and non-numeric inputs, which violates our sixth criterion. The programs taking numeric inputs in IntroClass have very simple functions (e.g. return the smallest of three numbers), which does not satisfy the first criterion.

4.5.2 Setup and Evaluation

First, we selected a classification algorithm (Section 4.4.2) for LEARN2FIX to train automatic oracles with. For each program subject, we applied LEARN2FIX selecting a random failing test case from the human-labelled test suite. In oracle learning, we simulate the human oracle ($\mathcal{H}$) as in Figure 3.3. After the automatic oracle ($\mathcal{O}$) is generated, we applied it to predict the labels of the test cases in the human-labelled test suite.

For the experiments, we fixed the following values.

- *Timeouts*: We allocated 10 minutes for oracle learning (Algorithm 1)
- *Committee size*: We set the size of the oracle committee to 20 members (i.e., $S = 10$ in Algorithm 2).
- *Maximum labelling effort:* We set the maximum number of queries presented to the human oracle ($\mathcal{H}$) as 20. (i.e., $L = 20$ in Algorithm 1)

Comparing the labels predicted by the automatic oracle ($\mathcal{O}$) with the actual labels of the test cases, we measure the following to evaluate the oracle quality.

i. Accuracy (Equation 3.1)

ii. Conditional accuracy - failing (recall for failing test cases) (Equation 3.2)

iii. Conditional accuracy - passing (recall for passing test cases) (Equation 3.3)

iv. Precision - failing (Equation 3.4)

v. Precision - passing (Equation 3.5)

We have identified that the labelled test suites of most selected subjects from Codeflaws have many passing test cases and few failing test cases. Thus, the *class imbalance problem* [17] affects the evaluation. For this reason, *accuracy* is not a good metric of oracle quality. As an example, an oracle predicting everything as passing would give 90% accuracy for a test suite containing test cases that are 90% *passing*. Therefore, we report *accuracy*, *conditional accuracy-failing*, *conditional accuracy-passing*, *precision-failing* and *precision-passing*. Also, these metrics help to evaluate the effectiveness of the techniques that LEARN2FIX uses to deal with the class imbalance problem (Section 4.4.1 & Section 4.4.3).

One objective of LEARN2FIX is to reduce the number of labelling queries presented to the human while maximising the probability of human labelling of failing test cases. To assess the success of the techniques used in LEARN2FIX to achieve this objective, we measure the following.

1. The proportion of generated tests that are labelled (Equation 4.2).
2. The proportion of failing tests that are labelled from the generated (Equation 4.3).
3. The probability to generate a failing test (Equation 4.4)
4. The probability to label a failing test (Equation 4.5).

Equation 4.4 indicates the probability of generating a failing test case in oracle learning. This is also the probability that the human would find a failing test only by mutational fuzzing and without LEARN2FIX. We compare this with the *probability of labelling a failing test case* (Equation 4.5). If the probability of labelling a failing test case is greater than the probability of generating a failing test case, the human needs less effort than usual to explore failing test cases. Equation 4.3 assesses the capability of LEARN2FIX to select failing test cases given by mutational fuzzing. By comparing this with the *proportion of generated tests that are labelled* (Equation 4.2), we can identify how effectively LEARN2FIX utilizes the given query budget to explore failing tests.

$$\begin{matrix} \textit{Proportion of generated} \\ \textit{tests labelled} \end{matrix} = \frac{\textit{Number of tests labelled}}{\textit{Number of tests generated}} \tag{4.2}$$

$$\begin{aligned} \textit{Proportion of failing} \\ \textit{tests labelled} \end{aligned} = \frac{\textit{Number of failing tests labelled}}{\textit{Number of failing tests generated}} \tag{4.3}$$

$$\begin{aligned} \textit{Probability of generating} \\ \textit{a failing test case} \end{aligned} = \frac{\textit{Number of failing tests generated}}{\textit{Total number of tests generated}} \tag{4.4}$$

$$\begin{aligned} \textit{Probability of labelling} \\ \textit{a failing test case} \end{aligned} = \frac{\textit{Number of labelled failing tests}}{\textit{Total number of labelled tests}} \tag{4.5}$$

To mitigate the impact of randomness and to gain statistical power for the experimental results, we repeated each experiment 30 times for each subject.

4.5.3 Implementation

All the experiments were implemented in Python 3.7. We used the *scikit-learn*[1] library to implement *decision tree*, *support vector machines*, *naïve bayes*, *AdaBoost* and *aritifical neural network* setups. INCAL has been implemented as a tool[2], which is supported by PYWMI[3], the LATTE[4] model counting tool and the NUMPY[5] scientific computing library.

4.6 Experimental Results and Discussion

To describe the *oracle quality* and *human labelling effort*, we use the results of LEARN2FIX under the *decision tree* algorithm. In determining the best category of classifier representation, we use the results of all the classification algorithms in Section 4.4.2.

4.6.1 Oracle Quality

Figure 4.3 shows the distributions in overall accuracy, conditional accuracy - failing, conditional accuracy - passing, precision-failing and precision - passing of the automatic

[1] https://scikit-learn.org
[2] https://github.com/ML-KULeuven/incal
[3] https://pypi.org/project/pywmi/
[4] https://www.math.ucdavis.edu/ latte/software.php
[5] https://numpy.org/

oracles generated by LEARN2FIX under the *decision tree* algorithm. For each subject, we computed the average of these metrics over the 30 runs.

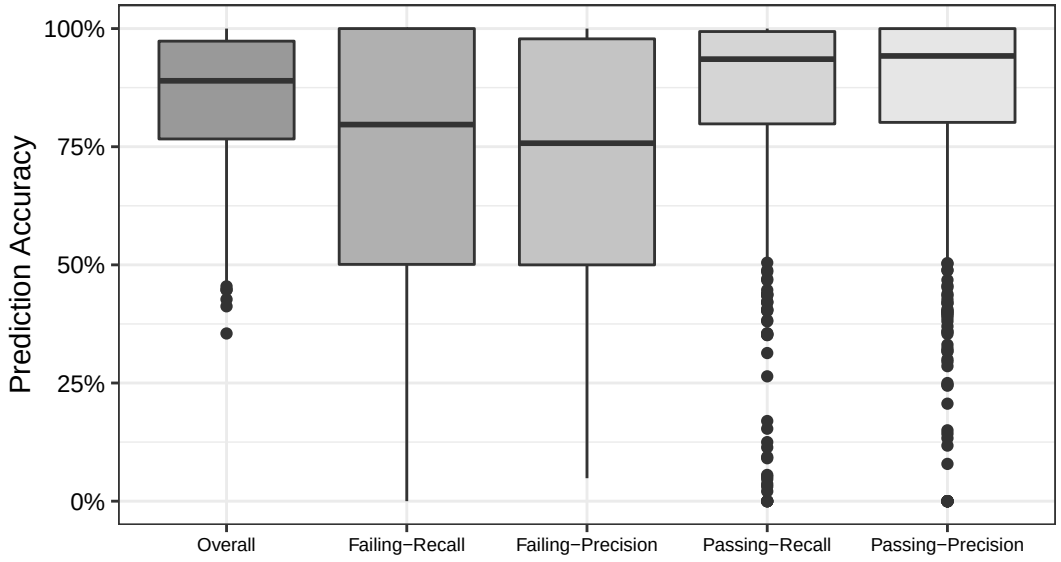
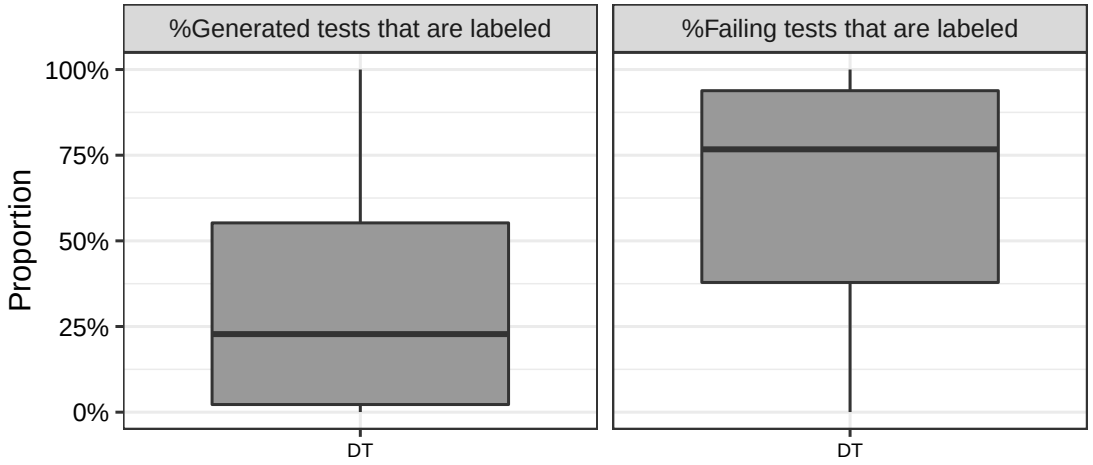


FIGURE 4.3: LEARN2FIX oracle quality under the *decision tree* algorithm

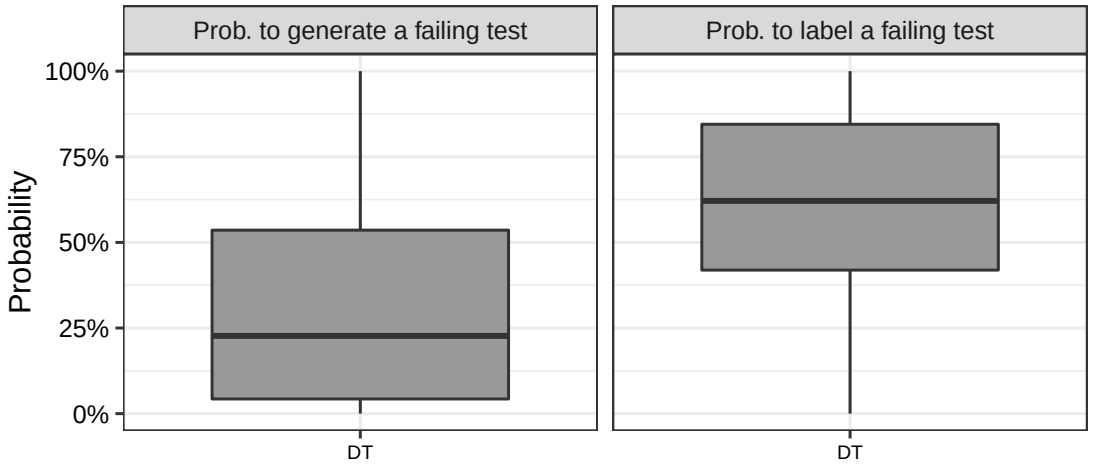
The automatic oracles trained by LEARN2FIX are able to accurately predict the labels of more than 89% of the manually labelled tests given by the benchmark of the majority of subjects. Even though LEARN2FIX saw only one failing test of the bug in the beginning, the automatic oracle correctly identifies more than 80% of the failing tests in most subjects. In addition, the precision and recall of passing test cases are above 90% for the median subject.

For all the metrics, the automatic test oracles trained by LEARN2FIX show higher median values ($> 75\%$). This indicates that even though the learning process begins with one failing test case, the automatic oracles for most subjects do not bias forward failing or passing test cases. Thus, the automatic oracles generated for the majority of subjects can accurately distinguish the passing and failing test cases.

Result. LEARN2FIX *generates high-quality automatic test oracles that can accurately identify passing and failing tests*



(a) Proportions of generated (left) / failing tests that are labelled (right)



(b) Probability to generate (left) / label a failing test (right)

FIGURE 4.4: LEARN2FIX human effort under *decision tree* algorithm

4.6.2 Human Labelling Effort

The boxplots in Figure 4.4(a) show the distributions of the proportion of generated tests that are labelled (left) and the proportion of generated failing tests that are labelled (right). Also, Figure 4.4(b) shows the distributions of the probability to generate a failing test (left) and the probability to label a failing test (right).

For the median subject, LEARN2FIX sends less than half of the generated failing test cases for human labelling. Also, LEARN2FIX sends more than 75% of the generated failing tests for human labelling (Figure 4.4(a)).
The median probability of labelling a failing test case (62.11%) is significantly higher than the median probability to generate a failing test (22.70%) (Figure 4.4(b)).

According to Figure 4.4(b)-left, only 25% of the generated tests are failing for the median subject. Thus, the human would not be able to label more failing tests by random labelling. Nevertheless, the higher median probability of labelling failing tests ($> 60\%$) indicates that LEARN2FIX significantly reduces the human effort to find a failing test case (Figure 4.4(b)-right). Also, Figure 4.4(a) indicates that most of the generated failing tests are sent for human labelling.

According to these results, the probability of generating failing test cases by *mutational fuzzing* is not at a significant level. Nevertheless, the DECIDE2LABEL algorithm has the ability to choose most of the generated failing tests by *mutational fuzzing*. As failing tests are the minority class, this capability of LEARN2FIX is advantageous to work with the *class imbalance problem* [17].

Result. LEARN2FIX *significantly reduces the human effort compared with random labelling*

4.6.3 Performance Under Different Classifier Representations

The boxplot in Figure 4.5 shows the distributions in the overall accuracy; Figure 4.6 distributions in the recall-failing (conditional accuracy - failing); Figure 4.7 distributions in the recall - passing (conditional accuracy - passing); Figure 4.8 distributions in the precision -failing and Figure 4.9 distributions in the precision - passing in each classification algorithm. Table 4.2 shows the mean and median values of each metric.

The boxplots in Figure 4.10 show the distributions in the metric used for assessing the human labelling effort. Table 4.3 shows the mean and median values of each metric.

For each subject, we computed the average of these metrics over the 30 runs under each classification algorithm.

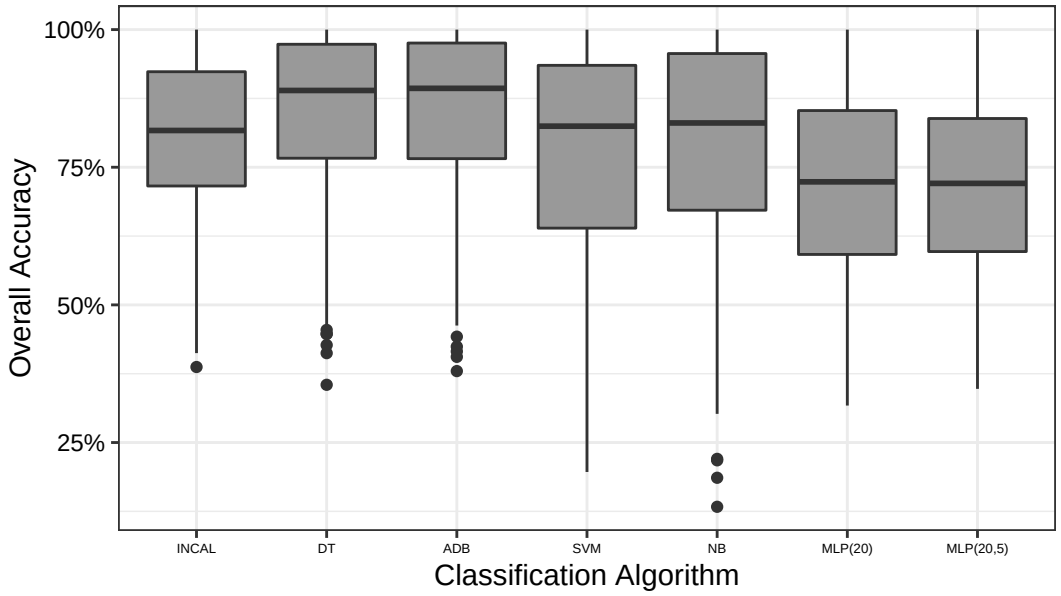


FIGURE 4.5: Overall accuracy under each classification algorithm

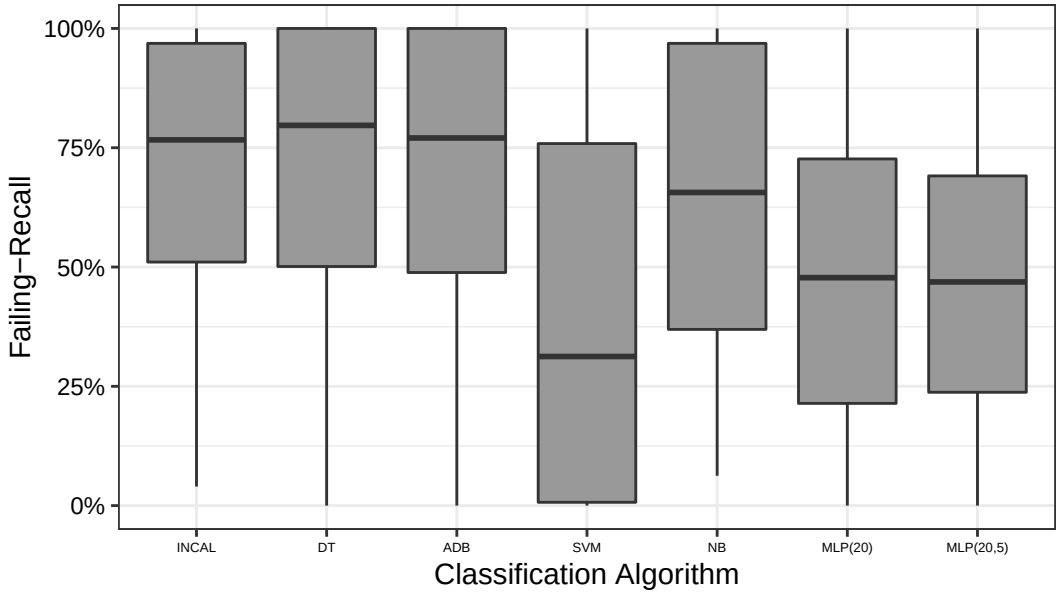


FIGURE 4.6: Recall-failing/conditional accuracy-failing under each classification algorithm

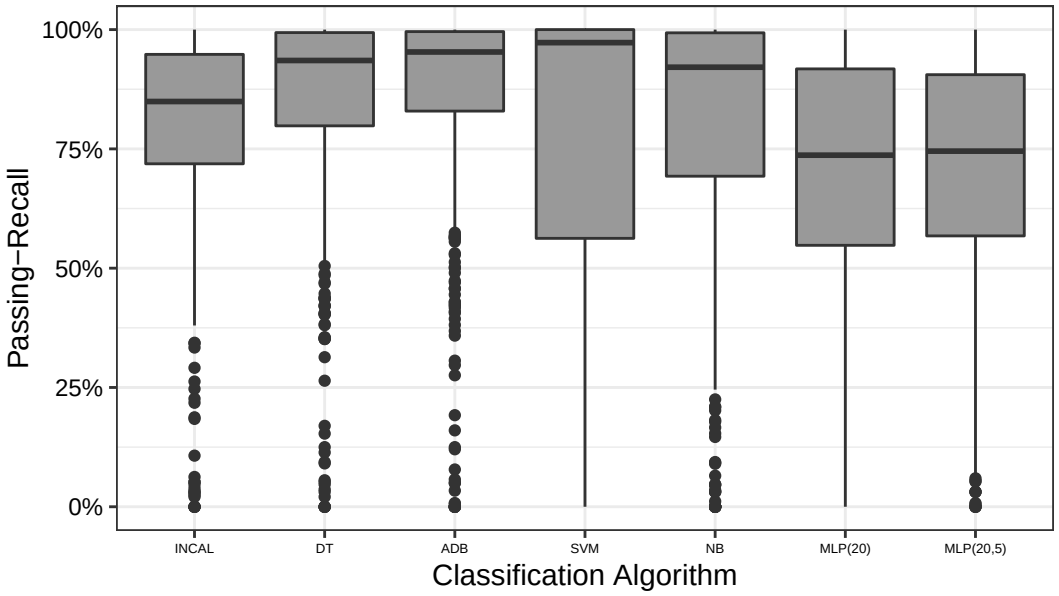


FIGURE 4.7: Recall-passing/conditional accuracy-passing under each classification algorithm

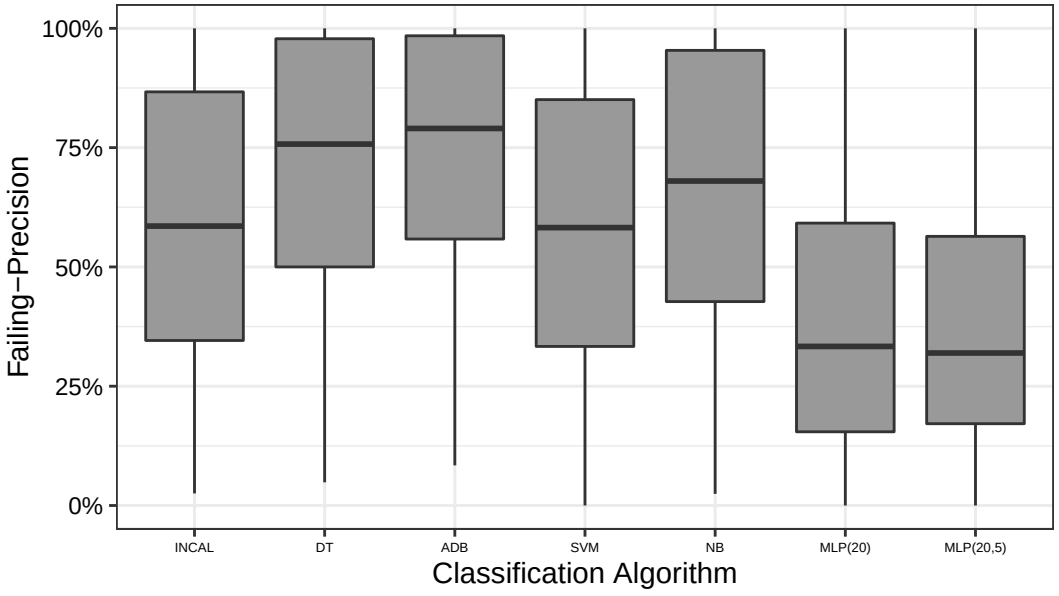


FIGURE 4.8: Precision-failing under each classification algorithm

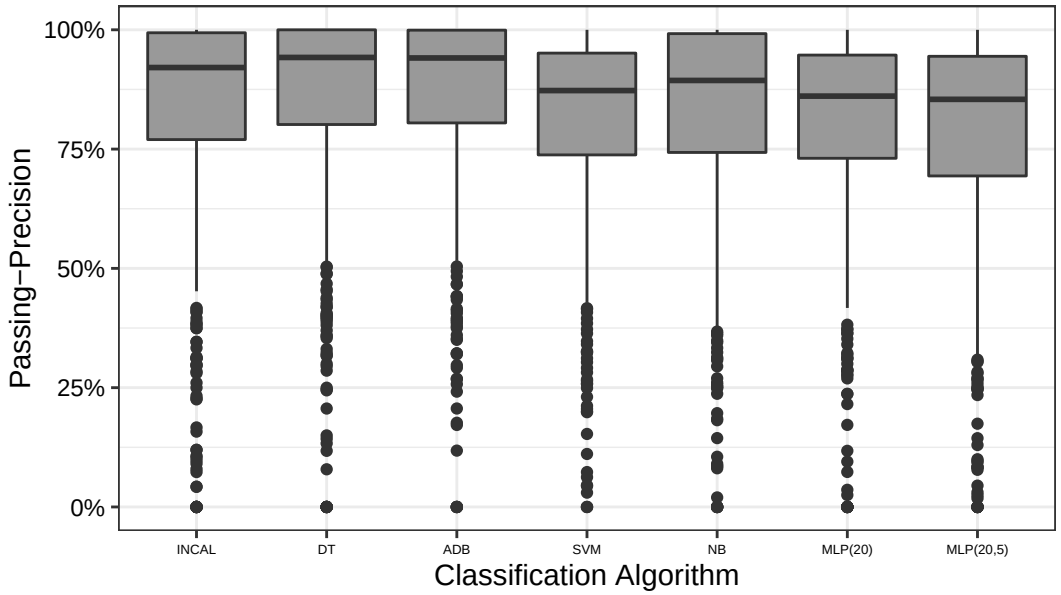


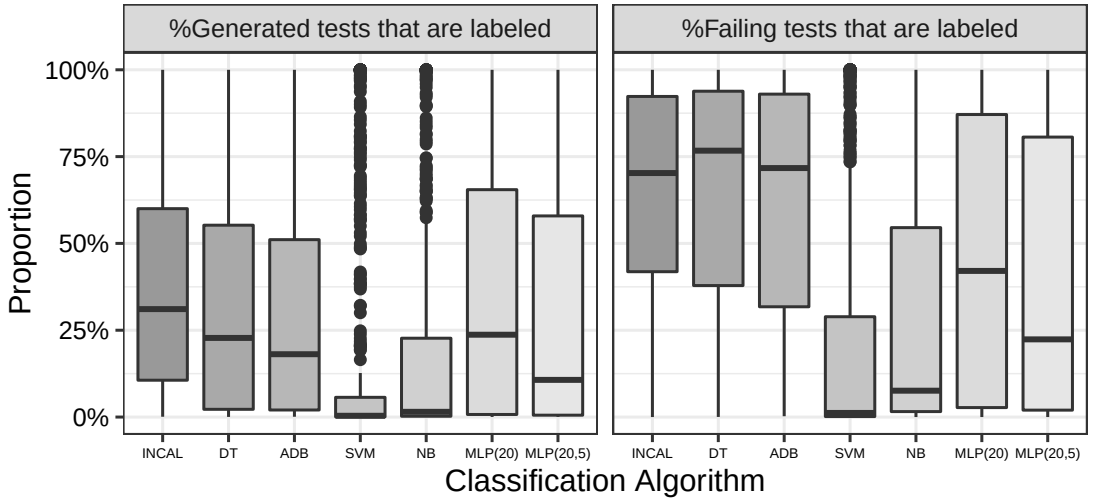
FIGURE 4.9: Precision-passing under each classification algorithm

Classification Algorithm	Overall Accuracy (%)		Recall Failing (%)		Precision Failing (%)		Recall Passing (%)		Precision Passing (%)	
	Mean	Median	Mean	Median	Mean	Median	Mean	Median	Mean	Median
Interpolation-based										
INCAL	80.74	81.68	71.68	76.65	59.30	58.56	79.29	84.93	82.80	92.09
Decision Tree	85.01	88.95	72.44	79.69	71.07	75.75	84.93	93.53	84.57	94.21
AdaBoost	85.38	89.34	70.64	77.05	74.02	79.01	85.87	95.31	85.25	94.10
Approximation-based										
SVM	77.70	82.46	39.51	31.25	58.79	58.24	77.51	97.27	80.41	87.27
Naïve Bayes	79.25	83.04	63.82	65.63	66.12	68	80.34	92.12	81.46	89.39
MLP (20)	72.43	72.35	48.15	47.77	39.67	33.33	70.67	73.68	79.09	86.10
MLP (20,5)	72.03	72.07	47.68	46.88	39.08	31.96	70.81	74.53	77.56	85.43

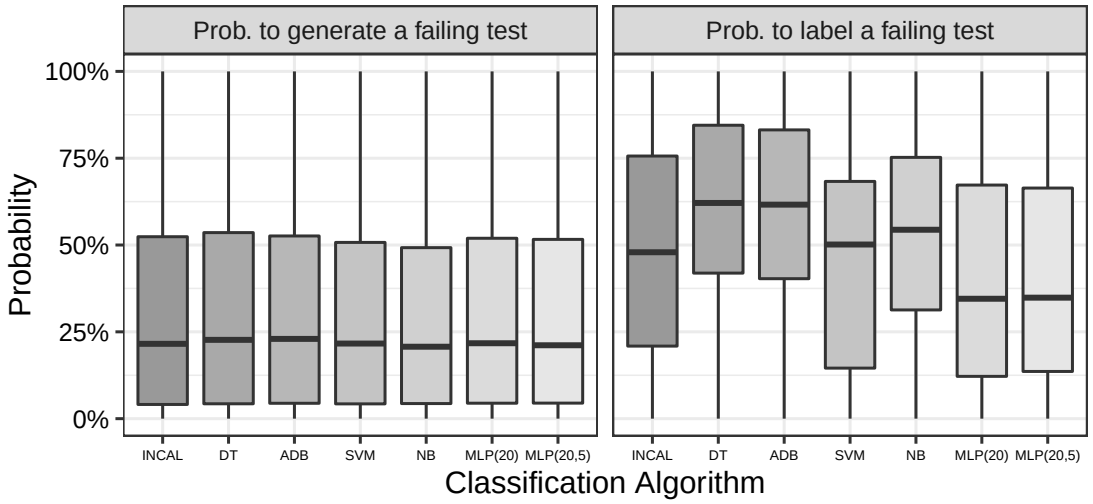
TABLE 4.2: Mean and median of the oracle quality of LEARN2FIX under different classification algorithms

LEARN2FIX trains better automatic oracles that identify test failures with the *interpolation-based* classification algorithms than with the *approximation-based* classification algorithms. The median conditional accuracy-failing (recall-failing) is above 75% in these algorithms. The *Decision tree* and *AdaBoost* algorithms generate the best automatic oracles with LEARN2FIX.

The Interpolation-based approaches work better with LEARN2FIX than the approximation-based approaches. Both *Decision Tree* and *AdaBoost* are better than *INCAL* in terms of oracle quality. According to the two-sided *Wilcoxon test*, the differences in the metrics



(a) Proportion



(b) Probability

FIGURE 4.10: Human effort under each classification algorithm

between *decision tree* and *AdaBoost* are statistically insignificant ($p > 0.05$).

In approximation-based approaches, only *Naïve Bayes* produces automatic oracles that identify test failures with significant accuracy ($> 60\%$) in most subjects. Even though *SVM*, *MLP (20)* and *MLP(20,5)* show more than 70% median overall accuracy, their median conditional accuracy - failing (recall - failing) is below 50%. According to the two-sided *Wilcoxon test*, the differences in the metrics between *MLP (20)* and *MLP(20,5)* are statistically insignificant ($p > 0.05$)

Classification Algorithm	Percentage of generated tests that are labelled		Percentage of Failing tests that are labelled		Probability to generate a failing test (%)		Probability to label a failing test (%)	
	Mean	Median	Mean	Median	Mean	Median	Mean	Median
Interpolation-based								
INCAL	36.92	31.09	64.13	70.26	30.20	21.55	48.08	47.92
Decision Tree	31.45	22.78	64.77	76.71	30.65	22.70	59.68	62.11
AdaBoost	29.64	18.1	62.42	71.70	30.60	22.97	59.13	61.63
Approximation-based								
SVM	17.33	0.38	22.26	1.19	30.26	21.63	45.63	50.15
Naïve Bayes	17. 53	1.51	28.80	7.6	29.58	20.72	53.49	54.40
MLP(20)	33.90	23.71	45.21	42.08	30.36	21.72	39.97	34.53
MLP(20,5)	28.89	10.70	38.95	22.37	30.36	21.14	40.07	34.84

TABLE 4.3: Mean and median values of the human effort of LEARN2FIX under different classification algorithm

Under all the classification algorithms, LEARN2FIX sends less than half ($< 50\%$) of the generated tests to the human in most subjects. The interpolation-based approaches *decision tree* and *AdaBoost* show around 60% median probability of labelling a failing test, which is approximately three times greater than finding a failing test by random labelling. (Table 4.3)

In the interpolation-based classification algorithms, more than 70% of the generated failing tests are sent to the human in most subjects. LEARN2FIX shows the highest median probability values for labelling a failing test under *AdaBoost* and *decision tree* algorithms. According to the two-sided *Wilcoxon test*, the differences in the metrics between *AdaBoost* and *decision tree* in the probability of labelling a failing test are insignificant ($p > 0.05$).

Even though the approximation-based approaches send only less than half of the generated tests, the percentage of generated failing tests sent for human labelling is significantly lower compared to the interpolation-based approaches. This result implies that the DECIDE2LABEL-algorithm works better with interpolation-based classification algorithms than approximation-based ones.

When considering both the human effort and oracle quality, the interpolation-based classification algorithms more effectively use the available human queries to improve the oracle quality than the approximation-based approaches. Even though the approximation-based approaches *SVM* and *naïve bayes* show around 50% median probabilities for

labelling a failing test, these algorithms are not as capable as the interpolation-based approaches in learning automatic oracles.

Result. LEARN2FIX *works better with interpolation-based classifier representation than with approximation-based classifier representation. Moreover,* LEARN2FIX *shows the best performance with the decision tree and AdaBoost algorithms.*

4.6.4 Discussion

Given a program with a semantic bug and a single failing test, LEARN2FIX learns a bug oracle as a classifier, which becomes the automatic oracle. The learnt automatic oracle ($\mathcal{O}$) expresses the condition under which the bug is exposed (i.e., *failure condition* of the semantic bug) LEARN2FIX improves the overall oracle quality by improving the classifier's ability to correctly identify failing tests, the minority class. For this purpose, LEARN2FIX maximises human labelling of failing tests in the learning process. The results of oracle quality (Section 4.6.1) and human labelling effort (Section 4.6.2) suggest that LEARN2FIX's oracle learning strategy works for many real-world semantic bugs in programs taking numeric inputs. The automatic oracles show more than 75% accuracy in identifying both passing and failing tests for most subjects. Manually exploring the failing tests of a semantic bug is a difficult task in programs taking numeric inputs. LEARN2FIX effectively addresses this issue via DECIDE2LABEL-algorithm. With the DECIDE2LABEL-algorithm, the probability of finding a failing test is *three* times higher than with random labelling.

In our experiments, we set a maximum limit (20) for the human labelling queries (Section 4.5.2). Nevertheless, the automatic oracles are highly accurate for most subjects (Section 4.6.1). This implies that LEARN2FIX's oracle learning strategy can maximise the oracle quality under the given limited human queries. LEARN2FIX achieves this capability with the help of all its components (4.1). This ability of LEARN2FIX is advantageous when the human (user or the developer) is the only source to obtain training data.

The results in Section 4.6.3 suggest that interpolating binary classifiers produce better automatic oracles than approximating binary classifiers with LEARN2FIX. This implies

that interpolating binary classifiers can better represent the failure condition of a semantic bug. Using *mutational fuzzing*, LEARN2FIX generates new test cases in the neighbourhood of the given failing test case. According to this result, finding a model that exactly fits the given failing test case and its neighbourhood test cases (i.e., interpolating a model) is the best strategy for learning an automatic oracle for the bug.

The results of Section 4.6.1 and Section 4.6.2 indicate that the ability of the classification algorithm to correctly capture the failure condition of a semantic bug determines:

i. The oracle quality, and

ii. The number of failing test cases sent for human labelling

If the classification algorithm can accurately capture the failure condition, the automatic oracle ($\mathcal{O}$) becomes accurate in identifying the failing test cases of the bug. As failing test cases are the minority class (Section 4.3), their accurate identification leads to higher oracle quality. Also, a classification algorithm capable of accurately capturing the failure condition sends a greater percentage of failing inputs generated in the learning process for human labelling.

INCAL produces classifiers as SMT constraints [129]. Also, a *decision tree* represents a set of constraints as a tree structure. *AdaBoost* is an ensemble version of decision trees; i.e., it combines several single-node decision trees to develop a classifier [179]. The median overall accuracy and conditional accuracy-failing of these algorithms are more than 70% (Table 4.2). Based on the statistical significance of the metrics, *decision tree* and *AdaBoost* perform equally well with LEARN2FIX and perform better than *INCAL*. According to the results, the overall accuracy, precision-failing and recall-passing of *INCAL* are significantly lower than those of these two algorithms. As *decision tree* and *AdaBoost* algorithms generate classifiers as decision trees, we conclude that *decision tree representation* is the best representation for the failure condition of a semantic bug in programs taking numeric inputs.

The lower conditional accuracy-failing of *SVM*, *MLP(20)* and *MLP(20,5)* implies that these algorithms are incapable of correctly learning the failure condition of a semantic bug. *Support vector machines* learn a linear discriminate function to separate data into classes [180]. *Artificial neural networks* infer a set of functions for the same task [181].

Finding a linear function or a set of functions to divide the passing and failing test cases of a semantic bug can be difficult. The reason is that the distributions of failing test cases of many semantic bugs are not continuous.

Naïve bayes has moderate performance according to the results in Table 4.2. This algorithm predicts the label of a data point based on the probability distribution of training data over classes. *Naïve bayes* considers the features of the training data in calculating the probability distribution [182]. The results (Table 4.2) indicate that this approach can accurately learn the failure condition of a semantic bug to a certain extent (median conditional accuracy-failing $\geq 50\%$); however, it is not effective as using constraints.

Under all the classification algorithms, the human has to label less than 50% of the generated test cases in the majority of the subjects (Figure 4.10(a) - *left*). However, only *decision Tree*, *AdaBoost* and *INCAL* maximise human labelling of failing test cases from generated failing test cases (Figure 4.10(a) - *right*). This occurs because these algorithms improve the accuracy of the automatic oracle ($\mathcal{O}$) as more failing test cases are given. This property is not seen in SVM, MLP(20) and MLP(20,5); hence, the lower conditional accuracy-failing (Table 4.2) and lower percentage of labelled failing tests from the generated (Table 4.3). According to the results, LEARN2FIX obtains over 60% median conditional accuracy-failing in naïve bayes, even though a very low percentage (7.6%) of generated failing tests are labelled. This implies that naïve bayes is able to learn the failure condition of a semantic bug using fewer failing test cases with reasonable ($> 50\%$) accuracy. However, it cannot improve the automatic oracle by obtaining more failing tests.

In all these algorithms, the median probability of labelling a failing test is higher than the median probability of generating a failing test (Table 4.3). This result indicates that Algorithm 2 in LEARN2FIX maximises the probability of labelling failing tests with all of the classification algorithms. Improving this probability reduces the difficulty of finding the failing tests of a semantic bug.

LEARN2FIX uses the techniques in Sections 4.4.1 and 4.4.3 to train automatic test oracles, dealing with the *class imbalance problem*. The key objective of these techniques is to improve the oracle's accuracy by maximising human labelling of failing test cases. According to the results, Algorithm 2 (DECIDE2LABEL) performs this task with all of the classification algorithms. However, only *decision Tree*, *AdaBoost* and *INCAL* can

produce high-quality automatic test oracles by labelling most failing test cases generated in the learning process. *Decision tree* and *AdaBoost* are the best among these algorithms.

4.7 Adversarial Learning to Improve Test Oracle Quality

We observed that LEARN2FIX is capable of maximising the oracle quality under the given limited human queries (Section 4.6.4). This is advantageous, as the training data for oracle learning is obtained from the human. To enhance the user-friendliness of LEARN2FIX, we explored methods for improving further the oracle quality under limited labelling queries. *Adversarial machine learning* [183] is a branch of machine learning that identifies the data misclassified by a machine learning model and uses those to improve the model. LEARN2FIX already has a method of maximising the human labelling of failing tests. We expected further to improve the oracle quality of LEARN2FIX by combining the capabilities of adversarial machine learning with our learning setup. To the best of our knowledge, adversarial machine learning has not been applied to learn test oracles.

Adversarial learning setups have been applied to improve fuzzing. The works of Yue et al. [184] and Woo et al. [185] are some examples. These two approaches use *multi-armed bandit (MAB)* algorithms [186] to implement the adversarial learning environments. *EcoFuzz* by Yue et al. [184] focuses on improving code coverage in *greybox* fuzzing [30]. Woo et al. [185] concentrate on increasing the number of bugs explored in a black-box fuzzing campaign. *EcoFuzz* considers the seeds as the arms of the multi-armed bandit setup. The number of seeds increases as the fuzzing process proceeds. *EcoFuzz* presents a multi-armed bandit approach to facilitate the *exploitation and exploration* in a setup where the number of seeds increases. This approach prioritizes the unfuzzed (newly added) seeds over already fuzzed seeds in the mutations. The reason is that it is impossible to explore the ability to reveal new paths of an unfuzzed seed without fuzzing it. Also, it assigns a reward to each fuzzed seed based on the number of new paths it has revealed to maintain a prioritization order among already fuzzed seeds.

Mutational fuzzing is the test generation component in LEARN2FIX (Table 4.1). A seed for the mutational fuzzing is randomly selected from the labelled failing tests (Algorithms 1 - Line 5 & 6). Following *EcoFuzz* [184], we introduced a biased seed selection

method to mutational fuzzing to generate test cases misclassified by the automatic oracle ($\mathcal{O}$) being trained. By human-labelling and using such test cases in training, we expected to achieve higher oracle quality under limited labelling queries. In LEARN2FIX, new seeds are added as the learning process proceeds. Hence, following *EcoFuzz*, we developed the two adversarial multi-armed bandit algorithms to achieve higher oracle quality under limited labelling queries. In these two algorithms, *mutational fuzzer* and the *automatic oracle* being trained are the two adversaries.

ADV-1. Average-sore based multi-armed bandit algorithm (Algorithm 4)

ADV-2. Page-Hinkley test based dynamic multi-armed bandit algorithm (Algorithm 5)

There are *static* and *dynamic* multi-armed bandit algorithms [186, 187]. Both types of multi-armed bandit algorithms can be applied to adversarial learning. In a static multi-armed bandit environment, the rewards associated with arms keep updating, and no resets happen in the middle. In contrast, a dynamic multi-armed bandit algorithm resets the rewards in the middle of the learning process. We developed **ADV-1.** as a static MAB algorithm and **ADV-2.** as a dynamic MAB algorithm to study which type of MAB is suitable for our setup.

Each adversarial multi-armed bandit algorithm has a BAISED_SELECT function and UPDATE_SCORE function. According to these two algorithms, we changed the LEARN2FIX oracle learning algorithm as in Algorithm 3. Each test case in $T_{\text{✗}}$ has a score. The BAISED_SELECT function (Line 5) selects a test case as a seed for the mutational based on the scores. Following *EcoFuzz* [184], it prioritizes newly added test cases in **ADV-1.** and **ADV-2.**. The objective of the DECIDE2LABEL function is to maximise the human-labelling of failing tests. If the DECIDE2LABEL function returns `true` for a passing test case, it signals that the automatic oracle ($\mathcal{O}$) has not been trained correctly (Section 4.4.3). Thus, the passing test case reveals an error of $\mathcal{O}$. In this case, the seed failing test (f') case generated this passing test case receives a reward (Line 13), as it (f') reveals an error of $\mathcal{O}$. The UPDATE_SCORE function updates the score of f'.

Algorithm 4 shows the process of **ADV-1.**. This is a *static multi-armed bandit* algorithm. It's BIASED_SELECT function performs the seed selection. As described before, if there is any failing test case in $T_{\text{✗}}$ that has not been fuzzed, it is selected as the

seed for the mutational fuzzing (Line 6-8). Unless, the BIASED_SELECT function calculates the average score (W_f) each failing test case in $T_{\text{✗}}$ (Line 10). This multi-armed bandit algorithm provides more opportunities to the exploration than *epsilon-greedy* algorithm [186]. Firstly, a newly added failing test case to $T_{\text{✗}}$ is selected in the next iteration for the mutation. Also, W_f decreases as f's ability to generate passing test cases for which the DECIDE2LABEL function returns `true`. The *update_score* function updates the score (S_f) associated with the selected seed failing test case. If $reward = 1$, S_f is increased; otherwise, no change happens.

Algorithm 5 shows the process of **ADV-2.**. This is a *dynamic multi-armed bandit* algorithm that has been influenced by the work of Luis et al. [187]. Luis's work develops a *dynamic multi-armed bandit (D-MAB)* environment by combining the *upper confidence bound*(UCB)-I [186] algorithm with the statistical test called *Page-Hinkley Test (PH-test)* [188]. As in **ADV-1.**, the BIASED_SELECT function of **ADV-2.** prioritizes the unfuzzed failing test cases in $T_{\text{✗}}$ as the seed for the mutational fuzzing (Line 6-7). If each failing test case in $T_{\text{✗}}$ has been fuzzed at least once, the BIASED_SELECT method applies the UCB-I algorithm. For each $f \in T_{\text{✗}}$, UCB-I algorithm calculates $d = \hat{r_f} + \sqrt{\frac{2 \log \sum_k n_k}{n_f}}$ (Line 9). The failing test case with the maximum d_f is selected as the seed for mutational fuzzing. In the UPDATE_SCORE function, m_f and M_f are the parameters of PH-test [188]. λ is a pre-defined threshold. If $M_f - m_f > \lambda$, the $n_f, \hat{r}_f, m_f$ and M_f of each failing test case in $T_{\text{✗}}$ are set to zero (Line 19-26). This is a resetting of rewards in all the seed-failing tests. This PH-test based part facilitates the dynamic multi-armed bandit environment.

As mentioned before, Algorithm 3 rewards the seed failing test cases that generate passing test cases for which DECIDE2LABEL function returns `true`. As the BIASED_SELECT function selects such seeds more frequently, Algorithm 3 maximises the generation of *passing* test cases having the following properties.

P-i. The current automatic oracle ($\mathcal{O}$) predicts as *failing*

P-ii. Having higher *failure likelihood* ($\hat{\theta} \geq 0.5$) according to the classifier committee in Algorithm 2)

These passing test cases signal the weakness of the oracle being trained ($\mathcal{O}$). Hence, human-labelling and using such passing test cases in training can rectify the automatic

oracle $\mathcal{O}$. In this manner, Algorithm 3 maintains the adversarial learning environment in oracle learning.

Algorithm 3 LEARN2FIX Active Oracle Learning with Adversarial Learning

Input: Buggy program ($\mathcal{P}$), Failing test case ($f = \langle \vec{i}, o \rangle$)
Input: Human oracle ($\mathcal{H}$), Maximum labelling queries (L)
Failing test cases $T_{\text{✗}} \leftarrow \{f\}$
Labelled test cases $T \leftarrow \{f\}$
Automatic Oracle $\mathcal{O} \leftarrow$ TRAIN_CLASSIFIER(T)
while ($|T| < L$) and not timed out **do**
  Failing test case $f^{'} \leftarrow$ BIASED_SELECT($T_{\text{✗}}$)
  Generate test case $t \leftarrow$ MUTATE_FUZZ($f^{'}$)
  if DECIDE2LABEL($t, \mathcal{O}$) = `true` **then**
    Human label $h = \mathcal{H}(t)$
    if $h =$ *fail* **then**
      reward $= 0$
      Failing test cases $T_{\text{✗}} \leftarrow T_{\text{✗}} \cup \{t\}$
    else
      reward $= 1$
    end if
    UPDATE_SCORE($f^{'}$, *reward*)
    Labelled test cases $T \leftarrow T \cup \{t\}$
    Automatic Oracle $\mathcal{O} \leftarrow$ TRAIN_CLASSIFIER(T)
  end if
end while

4.7.1 Experimental Setup and Evaluation

To evaluate the performance of **ADV-1.** and **ADV-2.**, we repeated the same set of experiments used to explore oracle quality and human labelling effort in Section 4.5.

As *decision tree* was one of the algorithms with the best performance in terms of oracle quality and human labelling effort (Section 4.6.3), we selected it as the classification algorithm in LEARN2FIX in these experiments. Also, the same set of subjects from *Codeflaws* [177] was used. The rest of the setup and the evaluation metrics are similar to Section 4.5.2.

In **ADV-2.** (Algorithm 5), we set $\lambda = 5$ and $\delta = 0.15$ based on the findings of [187].

Algorithm 4 Average-score based multi-armed bandit algorithm

```
1: T✗ : Failing test cases
2: Function BIASED_SELECT (T✗)
3:    // S_f : Score of f
4:    // N_f : Number of times t has been selected for mutations
5:    for f ∈ T✗ do
6:       if N_f = 0 then
7:          N_f ← N_f + 1
8:          return f
9:       else
10:         W_f ← S_f / N_f
11:      end if
12:   end for
13:   i = argmax_{f∈T✗} (W_f)
14:   N_{f_i} ← N_{f_i} + 1
15:   return f_i
16: EndFuction

17: Function UPDATE_SCORE(f, reward)
18:   S_f = S_f + reward
19: EndFunction
```

4.7.2 Experimental Results and Discussion

We compare the *oracle quality* and *human labelling* effort of the original LEARN2FIX (LEARN2FIX-DCT) and the multi-armed bandit algorithms (**ADV-1.** and **ADV-2.**).

4.7.2.1 Oracle Quality

Table 4.4 compares the results of *oracle quality* between the original LEARN2FIX (LEARN2FIX-DCT) and the multi-armed bandit algorithms (**ADV-1.** and **ADV-2.**).

Algorithm	Overall Accuracy (%)		Recall Failing (%)		Precision Failing (%)		Recall Passing (%)		Precision Passing (%)	
	Mean	Median	Mean	Median	Mean	Median	Mean	Median	Mean	Median
LEARN2FIX - DCT	85.01	88.95	72.44	79.69	71.07	75.75	84.93	93.53	84.57	94.21
ADV-1.	84.17	87.82	73.08	82.51	69.33	72.68	82.97	92.31	84.98	92.31
ADV-2.	84.22	87.72	72.91	81.35	69.06	73.55	83.29	92.17	84.70	94.20

TABLE 4.4: Oracle quality of LEARN2FIX with the multi-armed bandit algorithms

Algorithm 5 Page-Hinkley Test Based Dynamic Multi-Armed Bandit Algorithm

$T_{\text{✗}}$: Failing test cases
Function BIASED_SELECT ($T_{\text{✗}}$)
  // d_f : Score of f
  // n_f : Number of times f has been selected for mutations
  for $f \in T_{\text{✗}}$ **do**
    if $n_f = 0$ **then**
      return f
    end if
    $d_f \leftarrow \hat{r_f} + \sqrt{\frac{2 \log \sum_k n_k}{n_f}}$
  end for
  $i = \underset{f \in T_{\text{✗}}}{\text{argmax}}(d_f)$
  return f_i
EndFuction

Function UPDATE_SCORE(f, *reward*)
  $\hat{r}_f \leftarrow \frac{1}{n_f+1}(n_f \hat{r}_f + reward)$
  $n_f \leftarrow n_f + 1$
  $m_f \leftarrow m_f + (\hat{r}_f - reward + \delta)$
  $M_f \leftarrow \max(M_f, m_f)$
  if $M_f - m_f > \lambda$ **then**
    for $t_i \in T_{\text{✗}}$ **do**
      $n_f \leftarrow 0$
      $\hat{r}_f \leftarrow 0$
      $m_f \leftarrow 0$
      $M_f \leftarrow 0$
    end for
  end if
EndFunction

According to Table 4.4, slight changes can be seen in all the metrics between the multi-armed bandit algorithms and LEARN2FIX-DCT. According to the two-sided *Wilcoxon test*, these differences are statistically insignificant ($p > 0.05$).

Moreover, slight differences can be seen in all the metrics between the multi-armed bandit algorithms. The two-sided *Wilcoxon test* reveals that the differences are also statistically insignificant ($p > 0.05$).

Result. There is no evidence that ***ADV-1.*** *or* ***ADV-2.*** *improves the oracle quality under the limited human queries.*

4.7.2.2 Human Effort

Table 4.5 compares the results of *human effort* between the original LEARN2FIX (LEARN2FIX-DCT) and the multi-armed bandit algorithms (**ADV-1.** and **ADV-2.**).

Algorithm	Percentage of generated tests labelled		Percentage of failing tests labelled		Probability to generate a failing test (%)		Probability to label a failing test (%)	
	Mean	Median	Mean	Median	Mean	Median	Mean	Median
LEARN2FIX - DCT	31.45	22.78	64.77	76.71	30.65	22.70	59.68	62.11
ADV-1.	31.83	21.36	63.74	75.38	31.47	22.40	60.11	62.86
ADV-2.	34.08	25.69	66.93	79.11	32.80	25.08	61.29	64.69

TABLE 4.5: Human effort of LEARN2FIX with the multi-armed bandit algorithms

According to Table 4.5, slight differences can be seen in all the metrics between the multi-armed bandit algorithms and LEARN2FIX-DCT. However, the two-sided *Wilcoxon test* suggests that these differences are statistically insignificant ($p > 0.05$).

The two-sided *Wilcoxon test* further shows that the differences between the two multi-armed bandit algorithms in all the metrics are statistically insignificant ($p > 0.05$).

Result. There is no evidence that ***ADV-1.*** *or* ***ADV-2.*** *further reduces the human labelling effort*

4.7.2.3 Discussion

The experimental results provide no evidence that either **ADV-1.** or **ADV-2.** improves the oracle quality. Also, none of these algorithms reduces the human labelling effort. The slight changes in the metrics are due to the randomness; i.e., they are statistically insignificant.

ADV-1. and **ADV-2.** use biased seed selection methods instead of random seed selection in LEARN2FIX. The results provide no evidence that these biased selection methods improve the probability of labelling failing tests. Thus, the adversarial learning environment does not reduce the human effort associated with the learning process.

4.8 Culprit Constraint-based Approach to Improving Test Oracle Quality

In LEARN2FIX and the approaches used in Section 4.7, we consider the SUT (buggy program) as a "black-box". As the next step, we explored the applicability of using *white-box testing* [30] concepts to improve oracle quality. In white-box concepts, some information from the interior of the program is taken into consideration.

Culprit Constraint [189] is a branch condition that leads to a failure. Thus, a culprit constraint can explain the reason for a failing test. Also, it can assist LEARN2FIX in learning the failure condition of the bug in oracle learning. The work of Pham et al. [189] obtains culprit constraints by comparing the *path conditions* of passing and failing test cases. A *path condition* is a logical formula that consists of branch conditions connected with conjunctions ($\wedge$).

Definition 4.1 (Culprit Constraint)**.** Assume that the path condition of the test case t is ψ_t, and the path condition of the failing test case f is:

$$\psi_f = b_1 \wedge b_2 \wedge \ldots b_i \wedge \ldots b_n$$

Also, Π is the set of all passing path conditions. The branch condition b_i is the culprit constraint if and only if $i-1$ is the maximum value of j $(0 \leq j < n)$ such that $\mathit{prefix}(j, \psi_f) = \mathit{prefix}(j, \psi_p)$ for all $p \in \Pi$.

We identified that a culprit constraint of a buggy program reveals some information about the failure condition of the semantic bug, which could be difficult to learn from test cases with a classification algorithm. In addition, we observed the following properties of culprit constraints related to semantic bugs.

i. There can be more than one culprit constraint for a semantic bug.

ii. There can be both passing and failing test cases that satisfy a culprit constraint.

Considering these facts, we developed Algorithm 6.

Algorithm 6 Culprit-Constraint based oracle learning

Input: Path conditions passing tests (Φ_P), Path conditions failing tests (Φ_F).
Input: Labelled Tests (T)
1: $C = get\_culprit\_constraints(\Phi_P, \Phi_F)$
2: Test case groups $G \leftarrow \emptyset$
3: **for** $c \in C$ **do**
4: $g_c \leftarrow \emptyset$
5: **for** $t \in T$ **do**
6: **if** $isSatisfy(c, t)$ **then**
7: $g_c \leftarrow g_c \cup \{t\}$
8: $T \leftarrow T \setminus \{t\}$
9: **end if**
10: **end for**
11: $G \leftarrow G \cup \{g_c\}$
12: **end for**
13: **if** $T \neq \emptyset$ **then**
14: $G_0 = \{T\}$
15: $G \leftarrow G \cup G_0$
16: **end if**
17: Oracle groups $\mathcal{G} \leftarrow \emptyset$
18: **for** $g \in G$ **do**
19: $\mathcal{O} \leftarrow$ TRAIN_CLASSIFIER(g)
20: $\mathcal{G} \leftarrow \mathcal{G} \cup \mathcal{O}$
21: **end for**

4.8.1 Methodology

Algorithm 6 collects the path constraints of all the human labelled test cases. These path constraints contain only the inputs of the buggy program. At the end of Algorithm 1, Algorithm 6 obtains culprit constraints, following the method of [189]. Next, for each culprit constraint, Algorithm 6 finds test cases satisfying it from T and creates a group (Line 7). If there are test cases not satisfying any culprit-constraint in C, Algorithm 6 creates another group from these test cases (Line 13-15). Finally, we train a classifier for each group in G (Line 19-20). Thus, $\mathcal{G}$ contains a classifier (an automatic oracle) for each culprit constraint in C. Each culprit constraint acts as a *top-level constraint* to the classifier.

Assume that classifier $\mathcal{O}_i$ corresponds to the culprit constraint c_i ($\mathcal{O}_i \in \mathcal{G}$ and $c_i \in C$). Also, if there are test cases that do not satisfy any culprit constraint, $\mathcal{O}'$ is the classifier trained with them (G_0).

- $C = \{c_1, c_2, \ldots, c_n\}$

- $\mathcal{G} = \{\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_n, \mathcal{O}'\}$

This new group of classifiers predicts the label of a test case (t) as follows.

- $\exists i \in 1, \ldots, n : (isSatisfy(c_i, t) \wedge \mathcal{O}_i(t) = Failing) \vee (\mathcal{O}'(t) = Failing) : Failing$
- $\forall i \in 1, \ldots, n : \neg(isSatisfy(c_i, t) \wedge \mathcal{O}_i(t) = Failing) \wedge (\mathcal{O}'(t) = Passing) : Passing$

4.8.2 Experimental Setup and Evaluation

To evaluate this *white-box* approach, we repeated only the experiments related to oracle quality in Section 4.5. The reason is that Algorithm 6 changes the classifier at the end of oracle learning and thus does not affect Algorithm 2 (DECIDE2LABEL). Therefore, Algorithm 6 does not affect the human labelling effort.

In these experiments, we chose *decision tree* as the classification algorithm of LEARN2FIX, as it was one of the best performing classification algorithm in terms of oracle quality and human labelling effort (Section 4.6.3). Also, we selected 100 subjects from *Codeflaws* [177] satisfying the criteria in Section 4.5.1. We had to reduce the number of subjects due to certain technical limitations and because culprit constraints were not generated in some subjects. The rest of the setup and the evaluation metrics are similar to Section 4.5.2.

4.8.3 Experimental Results and Discussion

4.8.3.1 Oracle Quality

Table 4.6 compares the results of *oracle quality* between the original LEARN2FIX (LEARN2FIX-DCT) and the culprit-constraint based approach (LEARN2FIX culprit-constraint)

> *precision-failing* and *recall-passing* decrease with the culprit-constraint-based approach. According to the one-sided *Wilcoxon Test*, these decreases are statistically significant ($p < 0.05$), while the variations in the other metrics are not.

Algorithm	Overall Accuracy (%)		Recall Failing (%)		Precision Failing (%)		Recall Passing (%)		Precision Passing (%)	
	Mean	Median	Mean	Median	Mean	Median	Mean	Median	Mean	Median
LEARN2FIX - DCT	84.77	87.49	66.17	69.01	66.92	63.81	88.23	94.52	88.11	95.42
LEARN2FIX culprit-constraint	83.11	86.07	71.58	76.89	57.98	53.19	84.24	89.75	89.18	96.20

TABLE 4.6: Oracle quality of LEARN2FIX with the culprit constraint-based approach

The significant decreases in *precision-failing* and *recall-passing* indicate that the automatic oracles given by the culprit-constraint-based approach tend to incorrectly predict passing tests as failing.

Result. *The experimental results provide no evidence that the culprit constraint-based approach improves the oracle quality under the limited human queries.*

4.8.3.2 Discussion

According to the results, the culprit-constraint-based modification reduces the ability of automatic oracles to correctly identify passing tests. The key reason is that some culprit constraints do not correctly describe the failing test case, and no passing test cases are found in the learning process to prove this. For this reason, the test cases satisfying such culprit constraints are incorrectly predicted as *failing*.

In the experiments, we found that culprit constraints were not generated under some semantic bugs, as passing and failing tests follow the same path. We excluded such program subjects in these experiments. However, not generating culprit constraints hinders the generalizability of this approach. Also, all of the passing path conditions should be explored to accurately derive the culprit constraints. However, given the limited number of test cases, exploring all the passing path conditions might be impossible.

4.9 Extending LEARN2FIX for Other Unstructured Inputs

In extending LEARN2FIX to other unstructured inputs, the main concern is the *domain* of the targeted inputs. The classification algorithm (Section 4.4.2) to train automatic

oracles should be compatible with the domain. Also, the mutational fuzzing component (Section 4.4.1) should be adjusted to generate the valid inputs with respect to the domain.

As described in Section 4.1, numeric inputs are the widely used unstructured inputs in computer programs. Therefore, LEARN2FIX can be applied to a large range of programs that take unstructured inputs to generate automatic test oracles.

4.10 Conclusions

The main conclusions of the experiments with LEARN2FIX can be summarized as follows.

1. *Oracle Quality and Human Labelling Effort*
    - LEARN2FIX produces high-quality automatic test oracles under a limited number of human queries.
    - LEARN2FIX maximises the human labelling of *failing tests* in the learning process.
2. *Performance under different categories of classifier representations*
    - LEARN2FIX performs better with interpolating binary classifiers than approximating binary classifiers.
3. *Best representation for the failure condition of a semantic bug*
    - *Decision tree* is the best classifier representation. According to the results, classification algorithms producing classifiers as one or more decision trees lead to high-quality automatic test oracles.
4. *Adversarial learning and culprit-constraint-based approaches*
    - The experimental results provide no evidence that these approaches improve the oracle quality under limited human queries.

LEARN2FIX (without adversarial learning and culprit-constraint-based modification) is able to generate high-quality automatic test oracles for semantic bugs in programs taking numeric inputs. This process systematically interacts with the human in the learning

process. LEARN2FIX can be extended as an oracle learning framework for unstructured inputs. Therefore, LEARN2FIX can answer to **RQ.1**. Also, it is a contribution under **C.1**.

Chapter 5

Learning Automatic Test Oracles for Structured Inputs

This chapter explores an answer to **RQ.2** by developing an oracle learning technique for semantic bugs in programs taking *structured inputs*. *String inputs* instantiate structured inputs. Therefore, we present an approach called GRAMMAR2FIX to learn automatic oracles for semantic bugs in string processing programs. Beginning from a single failing input, GRAMMAR2FIX infers a general condition that a string input should satisfy to expose the bug. This condition is expressed as a *regular grammar* [142]. When this grammar is used as an automatic oracle, the test inputs adhering to the grammar are predicted as *failing*; otherwise, they are *passing*. We name these automatic oracles *grammar oracles*.

5.1 Structured Inputs

A *structured input* is associated with an underlying structure. The structure determines the validity of a given input. Usually, this kind of structure contains some basic elements that constitute the inputs. These elements are the fundamentals of the structure's definition. Unlike in unstructured inputs, a domain is unnecessary to determine the validity of structured inputs. The structure can be independently used to determine the validity of an input. This category of inputs is used in computer programs.

Computer programs and file formats are examples of structured program input types. A valid input of either type follows certain *syntactic rules* that define its structure. For instance, a valid computer program follows a set of rules defined by a programming language. The programming language defines the syntactic rules that a program should conform. *String inputs* are the most widely used structured inputs in programs. Email addresses, bank account numbers and phone numbers are strings that follow specific structures. *Formal grammar* can be applied to explain such a structure. For example, the *regular expression* [142] "`[a-z0-9]+` domain.com" describes the structure of the email addresses "abc@domain.com", "oracle112@domain.com", etc. (`[a-z0-9]+` : One or more alpha-numeric characters).

Unlike numeric inputs, the pattern of a group structured inputs cannot be explained based on a set of constraints defined on a domain. Hence, the technique introduced in Chapter 4 (LEARN2FIX) does not work for learning automatic oracles from structured inputs.

5.2 Motivation

As LEARN2FIX does not support structured inputs, we focus on developing a different oracle learning framework for semantic bugs in programs taking structured inputs. We select string inputs, as those are widely used in programs.

We demonstrate the challenge of learning automatic oracles for string processing programs using the buggy program in Listing 5.1. This program has been written to count the vowels of a given string ('A','E','I','O','U','a','e','i','o','u').

This program has a bug in Line 3. The list `vowels` does not contain the vowel 'a'. Thus, the program does not count the 'a's in a string. The program returns incorrect outputs for all strings containing 'a's, which is a semantic bug.

```
1 def find_vowel_count(input_str):
2         n_vowels=0
3         vowels=['A','E','e','I','i','O','o','U','u'] #Bug - no 'a'
4         for c in input_str:
5                 if( c in vowels ):
6                         n_vowels+=1
7         return n_vowels
```

LISTING 5.1: A Buggy Python program for counting the vowels of a string

Assume that a user finds that this program fails under the input "coverage". This string input has four vowels; however, the program returns a vowel count of three, as it does not count 'a's. With only this failing input and without program analysis, it is challenging to identify why the program fails. Similarly, a single failing string is insufficient to identify and fix the bug automatically. Similar to programs taking numeric inputs (Section 4.2), more passing and failing inputs are required to successfully execute automated program repair (APR) and debugging techniques with string processing programs. Finding more passing and failing inputs is challenging, as this is a semantic bug and only a human can determine whether a test is passing or failing. Like in programs taking numeric inputs, developing an automatic test oracle for this buggy program can overcome this difficulty.

In numeric program inputs, classification algorithms can be applied to successfully learn the failure condition of a semantic bug. However, these algorithms cannot be applied to non-numeric data. Therefore, we need to use a different approach from classification algorithms to train automatic oracles for semantic bugs in string processing programs. Without relying on any internal information of the buggy program, the failure condition of a semantic bug can be given as a pattern describing the failing inputs. As an example, the failure condition of Listing 5.1 can be expressed as *"The inputs containing 'a' are failing"*. Such a pattern can be easily used to generate more passing and failing inputs of the bug. Moreover, the nature of the bug can be studied by learning the pattern of the failing inputs. Based on these facts, a pattern describing failing inputs is the best representation for the failure condition of a semantic bug in a string processing program. Therefore, such a pattern is most suitable as an automatic oracle for the bug.

The pattern of a set of strings can be expressed as a *formal grammar* [137]. Thus, a formal grammar describing the pattern of failing inputs can be used an automatic oracle

for the bug. Therefore, we focus on *grammar inference* techniques to learn automatic oracles for semantic bugs in string processing programs based on passing and failing inputs. In developing this technique, we assume that a string processing program can accept almost any string (Similar to Listing 5.1).

5.3 Background

Strings or character sequences are another widely used type of program input. Usually, string processing programs accept almost any string (e.g., Listing 5.1). For a semantic bug in this category of programs, our objective is to develop an automatic test oracle as a grammar describing the pattern of the failing inputs. This grammar explains the failure condition of the bug in terms of the program inputs. Similar to the approach in Chapter 4, we assume that there are one failing input of the bug and the human (the user or the developer) to query the label of a test case.

In Chapter 4, we identified some constraint learning approaches for numeric inputs (e.g. INCAL [129]). However, to the best of our knowledge, there are no automated constraint learning approaches for strings, even though methods to solve string constraints have been discovered (e.g. Z3-str [190]). This is a motivation for focusing on grammar inference techniques in this oracle learning problem.

Similar to numeric inputs, one failing string input of the bug is insufficient to identify the reason behind the failure. Specific to string inputs, this task can be difficult, even when there are more randomly generated failing inputs. We identified several reasons for this difficulty.

Firstly, a string input can be interpreted in multiple ways. Thus, a failing string input could have multiple reasonable explanations for its failure. As an example, related to Listing 5.1, the input “coverage” might fail because:

i. It contains eight (8) characters.

ii. It starts with the character ‘c’.

iii. It ends with the character ‘e’. etc.

Secondly, a failing string input may contain character sequences that are optional to expose the bug. Removing such character sequences does not convert the failing input to a passing input. As an example, in the failing input "coverage" of Listing 5.1, the characters except 'a' are optional to expose the bug. Due to this reason, many semantic bugs in string processing programs have infinite failing inputs. Listing 5.1 is an example of such a buggy program, as there are infinite strings containing 'a'. Thus, the *class imbalance problem* [17] is not a critical consideration in string processing programs.

The difficulties described above make it challenging to infer a grammar describing the pattern of the failing inputs of a semantic bug. To address these issues, one strategy is to find the root cause of the failure first and guide the grammar inference process based on how the root cause appears in the failing inputs. This is a kind of *zoom in & zoom out* strategy [191].

Delta Debugging Minimization(ddmin) [77] is an algorithm applied to explore the root cause of a failure. *Iterative delta debugging* [95] and *hierarchical delta debugging* [96] are variants of this algorithm. *ddmin* minimizes the given failing input to the smallest input inducing the failure. New passing and failing inputs are intermediately generated in the minimization process. Thus, *ddmin* is suitable for our purpose.

Several algorithms have been proposed to infer grammar from examples. Regular Positive and Negative Inference (RPNI) [140], GOLD [141] algorithm and L^* [161] are some regular grammar inference algorithms. The Inductive CYK [143] is a context free grammar inference algorithm. Given a set of positive and negative examples, these algorithms infer a grammar that accepts the positive examples and rejects the negative ones. In addition to the examples, the alphabet of the target grammar should be predetermined in these algorithms. As the alphabet gets larger, the number of examples required for accurate grammar inference significantly increases. Similarly, more queries have to be sent to the oracle with active grammar inference algorithms (e.g., L^*). This is a significant limitation in applying these existing approaches to our problem. The reason is that the failing inputs of many semantic bugs in string processing programs have a large alphabet; therefore, a large number of examples should be human-labelled when inferring grammar by these approaches. This limitation is an important consideration in this oracle learning problem.

DDSET by Gopinath et al. [192] is an approach that explores the grammar of failing inputs, which is consistent with the objective of this chapter. However, DDSET assumes that the input grammar, i.e., the structure of valid program inputs, has been given. Usually, a program that processes strings accepts almost any string (e.g. program in Listing 5.1). The works of Baldoni et al. [175] and Bendrissou et al. [193] suggest that exploring the input grammar of such a program is challenging. This fact makes DDSET unsuitable for our task. Therefore, we concentrate on an oracle learning technique that is independent of the input grammar of a buggy program.

5.4 Methodology

With the buggy program ($\mathcal{P}$), GRAMMAR2FIX assumes that the following has been given.

1. One failing string input of the bug (f)
2. The human ($\mathcal{H}$) to answer whether a test is passing or failing

GRAMMAR2FIX follows a "zooming in and zooming out" approach to infer a grammar describing the pattern of the failing inputs. This approach has been applied in various domains, including research on decision-making and organizational studies [191]. The four main steps of GRAMMAR2FIX are as follows.

1. Minimise the given failing input (f) to the smallest failing input using *Delta Debugging Minimisation* [77].
2. Grammar Inference, which infers a grammar from the test inputs generated in the delta debugging minimisation (Step 1).
3. Grammar Generalization, which generalises the grammar using additional failing and passing inputs.
4. Grammar Extension, which extends the grammar created after Step 3 by finding more failing inputs through mutating the given failing input (f).

The first two steps zoom into the root cause of the failure and generate a first-level grammar. This grammar could be overfitting to a subset of failing inputs. Thus, the last

two steps zoom out by applying generalization and extension steps to avoid overfitting. These two steps involve some heuristics.

From *active learning* perspective, Step 1 (delta debugging minimization) and 3 (grammar generalization) follow *membership query synthesis* [123], i.e., these steps generate new string inputs and check whether they belong to the target grammar. Step 4 (grammar extension) follows *stream-based sampling*.

5.4.1 Delta Debugging Minimization (ddmin)

Delta Debugging Minimization(ddmin) [77] can reduce a failing input to a minimal failing input ($f_{\min}$) that reproduces the bug. A minimal failing input ($f_{\min}$) is a failing input that cannot be divided further to obtain more failing inputs. *ddmin* follows a *divide and conquer* approach to reduce the given failing input.

The *divide and conquer* approach of *ddmin* removes the parts that are not essential to exposing the bug associated with the failing input. Also, it divides the failing string input into smaller strings, which results in new test inputs. Thus, *ddmin* can be used as a test generation method.

Minimizing Delta Debugging Algorithm

$$ddmin(c_{\text{✗}}) = ddmin_2(c_{\text{✗}}, 2) \text{ where}$$

$$ddmin_2(c'_{\text{✗}}, n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{if } \exists i \in \{1 \ldots n\} \cdot test(\Delta_i) = \text{✗} \\ ddmin_2(\nabla_i, \max(n-1, 2)) & \text{else if } \exists i \in \{1 \ldots n\} \cdot test(\nabla_i) = \text{✗} \\ ddmin_2(c'_{\text{✗}}, \min(|c'_{\text{✗}}|, 2n)) & \text{else if } n < |c'_{\text{✗}}| \\ c'_{\text{✗}} & \text{otherwise} \end{cases}$$

where $\nabla_i = c'_{\text{✗}} - \Delta_i$, $c'_{\text{✗}} = \Delta_1 \cup \Delta_2 \cup \Delta_3 \ldots \Delta_n$, all Δ_i are pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx c'_{\text{✗}}/n$

The recursion invariant (and thus pre-condition) for $ddmin_2$ is $test(c'_{\text{✗}}) = \text{✗} \wedge n \leq |c'_{\text{✗}}|$.

GRAMMAR2FIX applies *ddmin* to the given failing input(f). The new test inputs generated intermediately are sent to the human oracle ($\mathcal{H}$) for labelling. Finally, *ddmin* returns the minimal failing input (f_{min}) of exposing the bug traced through f.

From *active learning* perspective, *ddmin* creates *membership queries* [123] to the human in the minimization. A membership query asks the human whether or not a test input is a member of failing inputs. Thus, the interaction between the human and *ddmin* is a *membership query synthesis* model.

Let F be the set of failing inputs, and P be the set of passing inputs, generated by *ddmin* for f. F and P are disjoint sets; i.e., $F \cap P = \emptyset$. Also, $f, f_{\text{min}} \in F$.

As an example, consider the buggy program in Listing 5.1 (Section 5.2). Table 5.1 shows the steps that *ddmin* follows to trace the minimal failing input (f_{min}) from the failing input "coverage".

Step	Test input		Test
1	Δ_1	cove	✓
2	Δ_2	rage	✗
3	Δ_1	ra	✗
4	Δ_1	r	✓
5	Δ_2	a	✗

TABLE 5.1: Applying *ddmin* to the failing input "coverage"

The outcomes of this process are as follows.

- $F = \{$"coverage", "rage", "ra", "a"$\}$
- $P = \{$"cove", "r"$\}$
- f_{min} = "a"

ddmin follows a systematic method (divide and conquer) to explore the minimal failing input of a given failing string input. Thus, the test inputs generated by *ddmin* illustrate the characteristics of the failure condition more concretely than randomly generated test inputs. By comparing the minimal failing input(f_{min}) with the intermediate test inputs $((F \setminus f_{\text{min}}) \cup P)$, we can identify how the minimal failing input can be extended to have more failing inputs. Thus, the intermediate passing (P) and failing inputs ($F \setminus f_{\text{min}}$) can be considered as the neighbourhood of the minimal failing input (f_{min}). Therefore,

a basic intuition about the failure condition of the bug can be developed based on the test inputs generated by *ddmin*. Hence, we use these test inputs in the next steps.

5.4.2 Grammar Inference (GI)

The objective of *grammar inference* (GI) is to infer a grammar that explains the pattern of the failing inputs generated by *ddmin*. To infer the grammar as a *Deterministic Finite Automata(DFA)*, we use the *Regular Positive and Negative Inference* (RPNI) [140] algorithm with a slight modification to its merging technique.

Definition 5.1 (Deterministic Finite Automaton (DFA))**.** A Deterministic Finite Automaton (DFA) can be defined by a 5-tuple $(Q, \Sigma, \delta, q_0, \mathcal{F})$ where Q is a finite set of states, Σ is a finite set of symbols called the alphabet, δ is the transition function: $\delta : Q \times \Sigma \rightarrow Q$, q_0 is the initial state ($q_0 \in Q$), and $\mathcal{F}$ is a set of final/accept states ($\mathcal{F} \subseteq Q$). A DFA can have self-transitions, i.e., transitions from one state to itself, and inter-state transitions, i.e., transitions between two states.

We selected DFAs to model the grammar due to the following reasons.

i. A DFA can be modelled to accept any given set of strings while rejecting all the other strings even without knowing the complete alphabet of the target grammar (e.g., a prefix tree acceptor, PTA).

ii. A DFA can be extended to accept more strings by adding more characters to the transitions.

iii. A collection of DFAs combined with disjunctions ($\vee$) can model more complex string patterns.

We selected (RPNI) to infer the DFA, as it is more accurate than the other DFA inference algorithms such as GOLD [141]. Also, it has a flexible merging technique that generalizes the DFA by merging the states.

Given a set of positive and negative examples, RPNI creates a DFA that accepts all the positive examples and rejects all the negative ones. As the grammar is developed for the failing inputs. we consider F as the set of *positive* examples and P as the set

of *negative* examples. According to the RPNI algorithm, first, we develop a *prefix tree acceptor* (PTA) based on the string inputs in F. This PTA is a DFA that accepts all the string inputs in F.

The next step of RPNI is to iteratively merge the possible states of the PTA such that no positive strings are accepted. In GRAMMAR2FIX, we introduce an additional constraint to merge the states as follows.

Definition 5.2 (Additional Constraint to RPNI merge)**.** Given a pair of states, if there are inter-state transitions that take any character of f_{min} (the minimal failing input), those states are not merged.

The characters of f_{min} are mandatory for the grammar of failing string inputs. The above constraint enforces the characters of f_{min} to appear only in the inter-state transitions of the DFA, which makes these characters mandatory in the grammar for failing inputs. In a DFA, the characters in a self-transition can appear zero or more times in a string accepted by the DFA at the specified position. Thus, the self-transitions indicate the characters at specific positions that are optional in exposing the failure. The DFA obtained in this step shows where and how f_{min} can appear in failing inputs.

As an example, consider the F obtained for the failing input "coverage" in Section 5.4.1. The PTA for all the failing inputs in F is given in Figure 5.1. The final state q_8 corresponds to "coverage"; q_{12} corresponds to "rage"; q_{10} corresponds to "ra" and q_{13} correpsonds to "a".

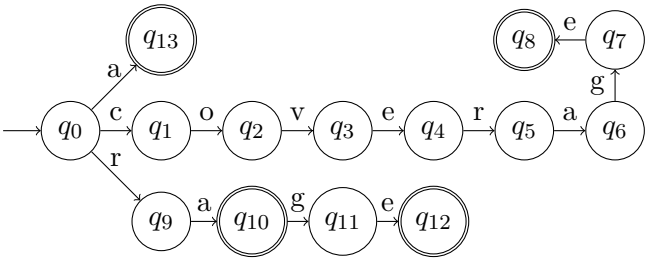


FIGURE 5.1: Prefix tree acceptor (PTA) for F derived from the failing input "coverage" of the buggy program in Listing 5.1

The inter-state transitions between q_0 and q_{13}, q_9 and q_{10} and q_5 and q_6 take 'a'. As 'a' is a character of f_{min}, the above state pairs are not merged according to Definition 5.2. The other states can be safely merged such that no string in P is accepted. The resulting DFA is given in Figure 5.2, which accepts all the strings in F. This DFA indicates that

“a” is mandatory in the failing inputs, and the other characters (‘c’,‘o’,‘v’,‘e’,‘r’,‘g’) can appear zero or more times.

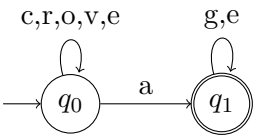


FIGURE 5.2: Basic level grammar with the modification to the RPNI merging technique. $f_{\min}$ =‘a’

The resulting grammar in this step is named *basic level grammar*.

5.4.3 Grammar Generalization

The *basic level grammar* is inferred based only on the test cases generated by *ddmin*. We can obtain only limited information about the failure condition from these test cases. Therefore, the basic level grammar can overfit the failing inputs in F. To avoid the overfitting, GRAMMAR2FIX applies the following generalization steps to the basic level grammar.

1. Basic Generalization (BG)
2. Handling Special Cases (HSC)
3. Finding the character class of the minimal failing input $f_{\min}$ (CCF)

BG focuses on generalizing the characters that are optional in exposing the bug. When the characters of $f_{\min}$ are placed at the beginning or end of f, HSC explores whether f can be extended to have more failing inputs of the bug. CCF explores more minimal failing inputs with the same length of $f_{\min}$.

5.4.3.1 Basic Generalization (BG)

There are few characters in the self-transitions of the basic level grammar (a DFA), as we used only the failing input set F (Section 5.4.1) to infer it. The characters that are mandatory in forming failing string inputs are already in the inter-state transitions of the DFA. Therefore, if there is a self-transition in a state, it can take any character

except those used in the outgoing inter-state transitions of the state. Based on this concept, we generalize the self-transitions of the states as follows.

Given a state q_i with one or more inter-state transitions,

i) BG identifies the set of characters (C_i) in the outgoing inter-state transitions of q_i.

ii) BG updates the characters of the self transition of q_i to include any character except the characters in C_i (${C_i}^{\mathrm{C}}$ - complement of C_i).

After the conversion, we name such self-transitions as *Complementary Self-Transitions*.

Example: Consider the DFA given in Figure 5.2. The state q_0 has an outgoing inter-state transition taking 'a' to reach q_1. Also, it has a self transition taking 'c', 'r', 'o' and 'v'. In this generalization, the self-transition is changed as it can happen under any character *except* 'a' ($\{\mathrm{a}\}^{\mathrm{C}}$), which is the *complementary self-transition* for q_0. State q_1 also has a self-transition that takes 'g' and 'e'. However, q_1 has no outgoing inter-state transitions. Thus, this generalization changes the self-transition in q_1 as it can happen under any character (because, $\emptyset^{\mathrm{C}} = U, U$ is the set of all characters). The DFA after this generalization is shown in Figure 5.3.

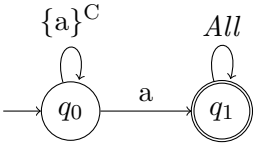


FIGURE 5.3: After adding complementary transitions to S and q_1. *All* is the set of all characters

5.4.3.2 Handling Special Cases (HSC)

We have identified that:

i. Positioning the characters of $f_{\min}$ at the *beginning* of failing inputs could lead to an *initial state* without complementary self-transitions.

ii. Positioning the characters of $f_{\min}$ at the *end* of failing inputs could lead to *final states* without complementary self-transitions.

These situations could either be attributes of the failing inputs or overfittings of them. To check this fact, we extend f (initial failing input), as *ddmin* cannot explore beyond the given failing input. *ddmin* avoids possible overfittings in the other states of the DFA, as it generates new test inputs through fragmenting f.

The HSC step considers the following properties as special cases, and GRAMMAR2FIX applies this step only if the DFA at this point exhibits these properties.

H-I. The initial state (q_0) has no complementary self-transition.

H-II. The final states have no complementary self-transitions and outgoing inter-state transitions.

If the DFA at this point has H-I., it could mean one of the following.

a) The failing inputs must start with the characters in the outgoing transitions of q_0

b) The current grammar overfits the given failing input (f).

To check which assumption is true, we create a random string *without* any character of f_{min} and add it to the front of f. The resulting test input is presented to the human. If this is a *failing* input, it signals the overfitting. Thus, we add a complementary transition to q_0.

Example: "apple" is a failing input of Listing 5.1. Also, $f_{\text{min=}}$ ='a'. The basic level grammar generated for this failing input would be as shown in Figure 5.4.

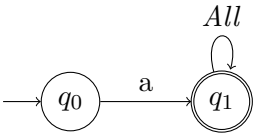


FIGURE 5.4: Basic level grammar for the input "apple", a failing input of Listing 5.1

The DFA in Figure 5.4 has no complementary self-transition in q_0 (the initial state). Following the process described above, we add a random string to the front of "apple" such that no character of $f_{\text{min=}}$ = is included. Assume that the new string is "Ykapple". This is a failing input, as this string contains 'a'. Thus, we add a complementary self-transition to q_0.

If the DFA at this point has H-II., it could mean one of the following.

a) The failing inputs must end at these states.

b) The current grammar overfits the given failing input (f).

To check which assumption is true, for each such final state, first, we select a failing test case that ends at the state from F (Section 5.4.1). A random string is added to the end of the selected failing input. The resulting input is presented to the human. If this is a *failing* input, it signals the overfitting. Thus, we add a complementary transition to the final state.

Example: "replica" is a failing input of Listing 5.1. Also, $f_{\text{min=}}$ ='a'. The basic level grammar generated for this failing input would be as shown in Figure 5.5

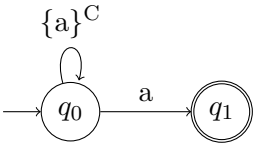


FIGURE 5.5: Basic level grammar for the input "replica", a failing input of Listing 5.1

The DFA in Figure 5.5 has no complementary self-transition in the final state q_1. "replica" ends at q_1. Following the process described above, we add a random string to the end of this string. Assume that the resulting string is "replicatxyq". This is a failing input, as this string contains 'a'. Thus, we add a complementary self-transition to q_1.

5.4.3.3 Finding the character class of the minimal failing input f_{min} (CCF)

We discovered that, under some bugs, there can be more than one minimal failing input of the same length. In such situations, more minimal inputs can be explored by substituting the characters of one minimal failing input. Thus, the *character class finding* (CCF) step focuses on finding the character substitutions, i.e., the character class, producing minimal failing inputs. Algorithm 7 shows the overall process used in the CCF step to find the character class of a given minimal failing input.

The CCF step follows the assumption given below.

Assumption: *The pattern of a group of minimal failing inputs of the same length can be abstracted in terms of the unique characters of one minimal failing input of the group.*

Algorithm 7 Character Class Finding

Input: $f_{\min}$: Minimal Failing Input
Input: $\mathcal{H}$: Human Oracle
Input: n : Substitution Iterations
Output: $C_{f_{\min}}$: Character Class of $f_{\min}$

1: $U \leftarrow unique\_characters(f_{\min})$
2: $P_A \leftarrow \emptyset$
3: $n\_success \leftarrow 0$
4: **for** $i \leftarrow 1$ to n **do**
5: $\quad \mathcal{N} \leftarrow get\_new\_random\_assignment()$
6: $\quad t_{new} \leftarrow replace\_unique\_characters(f_{\min}, \mathcal{N})$
7: $\quad$ **if** $\mathcal{H}(t_{new}) = Pass$ **then**
8: $\quad\quad P_A \leftarrow P_A \cup \{\mathcal{N}\}$
9: $\quad\quad n\_pass \leftarrow n\_pass + 1$
10: $\quad$ **end if**
11: **end for**
12: **if** $\frac{n\_pass}{n} = 1$ **then**
13: $\quad C_{f_{\min}} \leftarrow U$ {**Case 1**}
14: **else**
15: $\quad$ {Let All be the set of all possible character substitutions}
16: $\quad C_{f_{\min}} \leftarrow All \setminus P_A \cup \{U\}$ {**Case 2**}
17: **end if**
18: **return** $C_{f_{\min}}$

This assumption helps to reduce the search space when an input grammar is unavailable. Following this assumption, Algorithm 7 substitutes the unique characters of $f_{\min}$ with a set of random characters distinct from each other. In other words, if the set of unique characters of $f_{\min}$ is $U = \{C_1, C_2, \cdots, C_n\}$ ($C_1 \neq C_2 \neq C_3 \cdots \neq C_n$), Algorithm 7 substitutes $C_1 \leftarrow A_1$, $C_2 \leftarrow A_2$, $C_3 \leftarrow A_4$, $\cdots$, $C_n \leftarrow A_n$, where $\{A_1, A_2, A_3, \cdots A_n\}$ is the set of random characters and $A_1 \neq A_2 \neq A_3 \cdots \neq A_n$ (Line 5). The resulting test input is presented to the human ($\mathcal{H}$) for labelling. Through our experiments, we have identified that this technique can effectively explore minimal failing inputs of the same length.

Example: If $f_{\min}$="abab" , there are two unique characters 'a' and 'b', and we assume that the pattern of the minimal failing inputs are of the form $C_1C_2C_1C_2$ (where $C_1 \neq C_2$). Then, we substitute 'a' $\leftarrow$ 'c' and 'b' $\leftarrow$ 'd'. This substitution produces "cdcd", which is presented to the human. (Here, substitutions such as 'a' $\leftarrow$ 'c' and 'b' $\leftarrow$ 'c' are considered invalid, as 'a' and 'c' are substituted with the same character which breaks the pattern).

Algorithm 7 repeats this process for n iterations (Line 4). There are two main cases depending on the number of passing inputs generated in this process.

Case 1 All the generated inputs are *passing* . We conclude that f_{min} is the only minimal failing input, and the character class ($C_{f_{\text{min}}}$) is only the set of unique characters (U) (Line 13).

Case 2 Not all the generated inputs are *passing*. We conclude that except for the substitutions leading to passing inputs (P_A), the unique characters of f_{min} can be replaced by any other set of characters distinct from each other. Thus, $C_{f_{\text{min}}} = All \setminus P_A \cup \{U\}$ (Line 16)

In Algorithm 7, P_A and $C_{f_{\text{min}}}$ are sets of sets.

Under **Case 1**, no change is done to the DFA. The buggy program in (Listing 5.1) falls under **Case 1**, as the 'a' is the only minimal failing input.

Under **Case 2**, if $s \in C_{f_{\text{min}}}$, for each character in U in the inter-state transitions, there is a corresponding character in s. By replacing each character of the inter-state transitions with its corresponding character in s and changing the complementary self-transitions accordingly, a new DFA is created. This is done for all the sets in $C_{f_{\text{min}}}$, which results in a collection of DFAs that are connected with disjuctions/"OR ($\bigvee$)" operator.

Example: Consider f_{min}="abab", and the DFA presented in Figure 5.6. Assuming **Case 2** and Algorithm 7 returns the set of character substitutions:

$$C_{f_{\text{min}}} = \{\{C_{11}, C_{21}\}, \{C_{12}, C_{22}\} \cdots \{C_{1n}, C_{2n}\}\}$$

The DFA in Figure 5.6 is converted to a collection of DFAs connected with "OR" operators as in Figure 5.7, where C_{ij} is the assignment to the i$^{\text{th}}$ unique character of f_{min} from the j$^{\text{th}}$ substitution set.

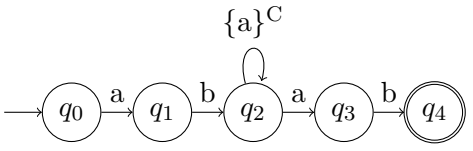


FIGURE 5.6: DFA before finding the character class of f_{min}

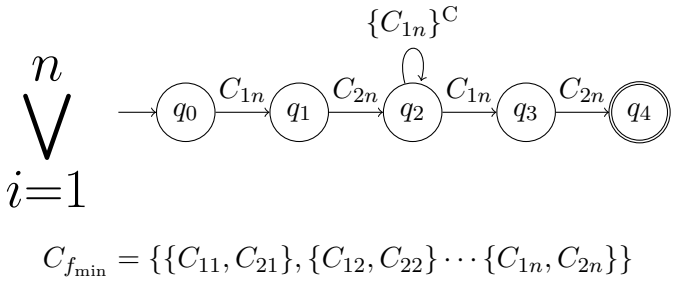


FIGURE 5.7: Abstract representation of the collection of DFAs

From *active learning* perspective, both HSC and CCF steps are *membership query synthesis* [123] models, as these steps create new test inputs and ask their labels.

5.4.4 Grammar Extension(GE)

One failing input can be insufficient for capturing all the properties of the failure condition for certain semantic bugs, even with the steps followed so far (Section 5.4.1 - Section 5.4.3.3). As an example, there are bugs with several *minimal failing* inputs of different lengths, which cannot all be explored by the CCF step. In addition, due to the limited number of substitution iterations, CCF might not find all minimal failing inputs. These issues can be addressed by exploring the neighbourhood of the given failing input (f).

The key objective of *grammar extension*(GE) is to extend the grammar oracle at this point by exploring the neighbourhood failing inputs of f. Algorithm 8 describes the overall process of GE.

To explore neighbourhood failing inputs of f, GE uses *mutational fuzzing* [29] with N fuzzing iterations. In each iteration, the newly generated test input (t_s) is presented to the current grammar oracle ($\mathcal{O}_\mathcal{G}$). If O_G predicts t_s as *passing*, i.e., t_s does not adhere to the inferred grammar at this point, t_s is sent to the human (Line 6 & 7). If the human labels t_s as *failing*, it implies that the current grammar oracle (O_G) cannot correctly identify this failing input. Furthermore, the grammar has not been trained to accurately identify the failure condition of the bug. Thus, GE applies step 1 to 3 (ddmin to CCF) of GRAMMAR2FIX (Line 8) to t_s and derives a new grammar ($\mathcal{G}_{\text{new}}$). Next, GE connects $\mathcal{G}_{\text{new}}$ to the grammar oracle ($\mathcal{O}_\mathcal{G}$) by "OR" operator (Line 9). Also, t_s is added to the seed corpus ($\mathcal{C}$). As test inputs are sampled for labelling from a stream of test

Algorithm 8 Grammar Extension

Input: f : Initial failing input
Input: $\mathcal{O}_\mathcal{G}$: Grammar oracle
Input: $\mathcal{H}$: Human oracle
Input: N : Fuzzing iterations

Let $\mathcal{C}$ be the seed corpus of failing inputs.
$\mathcal{C} \leftarrow \{f\}$
for $i \leftarrow 1$ to N **do**
  $f' \leftarrow pick\_random(\mathcal{C})$
  $t_s \leftarrow mutate\_fuzz(f')$
  if $\mathcal{O}_\mathcal{G}(t_s) = Pass$ **then**
    if $\mathcal{H}(t_s) = Fail$ **then**
      $\mathcal{G}_{\text{new}} \leftarrow Derive\_Grammar(t_s)$
      $\mathcal{O}_\mathcal{G} \leftarrow \mathcal{O}_\mathcal{G} \vee \mathcal{G}_{\text{new}}$
      $\mathcal{C} \leftarrow \mathcal{C} \cup \{t_s\}$
    end if
  end if
end for

cases generated by mutational fuzzing, GE step follows the *stream-based sampling* [123] in active learning.

At the end of the four steps (Sections 5.4.1/ddmin - 5.4.4/GE), GRAMMAR2FIX returns the grammar describing the failure condition, i.e., grammar oracle, as a DFA or a collection of DFAs connected with "OR" operators.

Figure 5.8 shows the overall workflows of the GRAMMAR2FIX grammar inference.

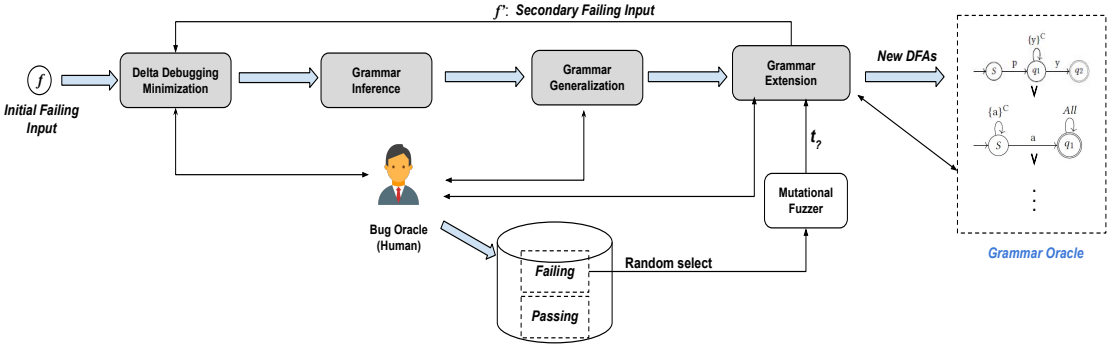


FIGURE 5.8: Workflow of GRAMMAR2FIX

The components of GRAMMAR2FIX can be mapped to the oracle learning in architecture in Figure 3.1 as in Table 5.2

Test Generation Technique	Mutational Fuzzing - Section 5.4.4
Active Learning Technique	• Membership query synthesis – ddmin - Section 5.4.1 – HSC - Section 5.4.3.2 – CCF - Section 5.4.3.3 • Stream-based sampling – GE - Section 5.4.4
Supervised Machine Learning Algorithm	Modified RPNI algorithm -Definition 5.2

TABLE 5.2: Mapping of GRAMMAR2FIX's components to the oracle learning architecture

5.5 Experimental Setup

We conducted several experiments to evaluate the performance of GRAMMAR2FIX in terms of the following aspects.

i. Oracle quality

ii. Human labelling effort

iii. The impact of the heuristics used in grammar inference (GI), basic generalization (BG), handling special cases (HSC), character class finding (CCF), and grammar extension (GE) (Sections 5.4.2 - 5.4.4) on the oracle quality and human labelling effort.

5.5.1 Experimental Subjects

We selected three *benchmarks* for the experiments according to the criteria given below .

1. There should be programs that take string inputs.

2. There should be a diverse set of real defects that lead to *functional bugs*; i.e., programs produce incorrect outputs for specific inputs. There should be one functional bug for each subject.

3. For each subject, there should be a *golden version*, i.e., a program that produces the expected, correct output for an input. For a given input, we simulate the human($\mathcal{H}$) by comparing the subject's (buggy program's) output with its golden version's output. If both are different, the test case is labelled as failing. (Figure 3.3)

4. For each subject, there should be a *manually constructed* and *labelled* test suite.

5. For each subject, there should be at least one *failing* test case in the human-labelled test suite, i.e., a test input for which the buggy and the golden version produce different outputs.

We found that the benchmarks *IntroClass* [178], *Quixbugs* [194], and *Codeflaws* [177] satisfy the above criteria. *IntroClass* consists of C programs that were submitted under six (6) assignments by a group of students. We selected the programs under the assignments *Syllables* and *Checksum*, as those take string inputs. Under each of the two assignments, there is a golden version and a labelled test suite. Based on the second and fifth criteria, we excluded the programs showing flaky behaviour and having no failing inputs. After that, there were 52 subjects under *checksum* and 121 subjects under *syllables* for the experiments.

We selected four (4) Python programs from *Quixbugs* [194] and 152 C programs from *Codeflaws* [177] based on the first criterion. In this selection, we excluded the programs that take mixed inputs (e.g., strings and numbers together) from the set of benchmarks. For each of these selected subjects, there is a separate labelled test suite and a golden version.

Benchmark	Language	Number of subjects
IntroClass	C	163
Codeflaws	C	152
QuixBugs	Python	4

TABLE 5.3: Subject selection for the GRAMMAR2FIX experiments

5.5.2 Setup and Evaluation

For each subject, we applied GRAMMAR2FIX by selecting a random failing input from the training test suite (manually labelled test suite). After the grammar oracle($\mathcal{O}_\mathcal{G}$) is

generated, we applied it on the training test suite. As described in the beginning, if a test input adheres to the grammar, it is predicted as *failing*; otherwise, it is *passing*.

For the experiments, we fixed the following values.

- *Timeouts*: We allocated 10 minutes to generate the grammar oracle for each subject.
- *Substitution iterations in CCF (Algorithm 7)*: 20 iterations ($n = 20$)
- *Fuzzing iterations in GE (Algorithm 8)*: 5 iterations ($N = 5$)

Comparing the predicted labels by the grammar oracle ($\mathcal{O}_{\mathcal{G}}$) with the actual labels of the test cases, we measure the following to evaluate the oracle quality.

i. Accuracy (Equation 3.1)

ii. Conditional Accuracy - Failing (Recall for failing test cases) (Equation 3.2)

iii. Conditional Accuracy - Passing (Recall for passing test cases) (Equation 3.3)

iv. Precision - Failing (Equation 3.4)

v. Precision - Passing (Equation 3.5)

Similar to Chapter 4, the labelled test suites provided by the benchmarks are imbalanced. Thus, the evaluation is affected by the *class imbalance problem* [17]. Therefore, we report the metrics Equation 3.1 - Equation 3.5. In addition, regarding the human labelling effort, we just report the number of queries sent to the human, as there is no maximum limit for the labelling queries.

To evaluate the impact of the heuristics used in GRAMMAR2FIX, we repeated the above experiment separately up to each generalization step; i.e., GI, GI to BG, GI to HSC, GI to CCF and GI to GE. In each case, we determine the metrics (Equation 3.1 - Equation 3.5) and the number of queries sent to the human.

To mitigate the impact of randomness and to gain statistical power for the experimental results, we repeated each experiment 30 times for each subject.

5.5.3 Implementation

GRAMMAR2FIX and all the experiments were implemented in Python 3.7.

5.6 Experimental Results and Discussion

5.6.1 Oracle Quality

Benchmark	Overall Accuracy (%)		Recall Failing (%)		Precision Failing (%)		Recall Passing (%)		Precision Passing (%)	
	Mean	Median	Mean	Median	Mean	Median	Mean	Median	Mean	Median
IntroClass	88.66	99.21	88.34	98.73	83.17	100	84.46	100	88.12	96.97
Codeflaws	72.47	77.70	73.13	86.88	61.22	58.99	67.45	78.92	79.70	91.49
QuixBugs	88.69	88.36	93.80	99.31	88.84	93.24	90.52	96.09	82.08	96.77

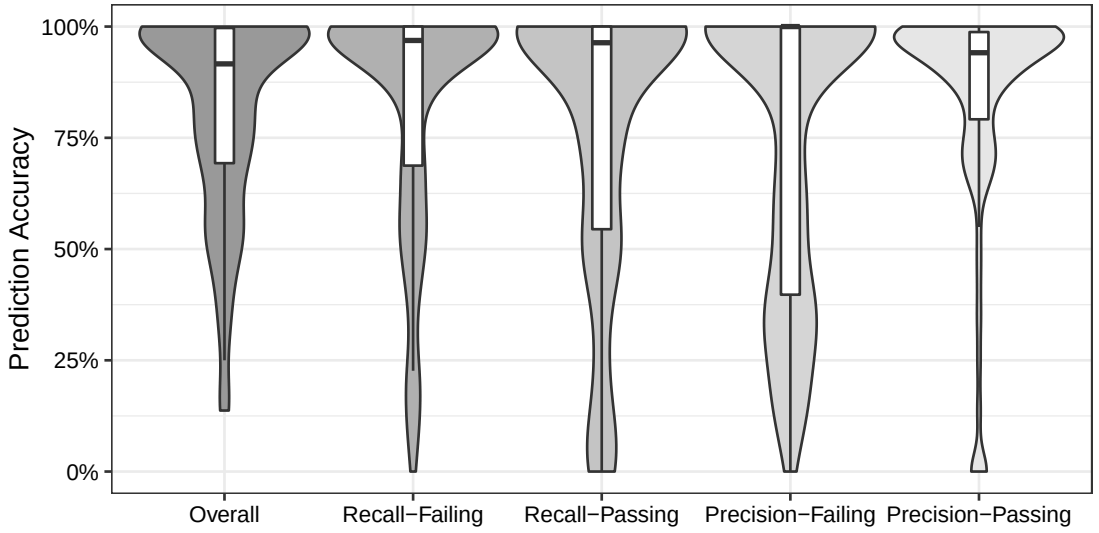
TABLE 5.4: Mean and median of the oracle quality of GRAMMAR2FIX under three benchmarks

Figure 5.9 shows the distributions in overall accuracy, conditional accuracy - failing, conditional accuracy - passing, precision-failing and precision - passing of the automatic oracles generated by GRAMMAR2FIX. Figure 5.9(b) shows the distributions of these metrics benchmark-wise. In addition, Table 5.4 gives the mean and median values of these metrics for each benchmark.

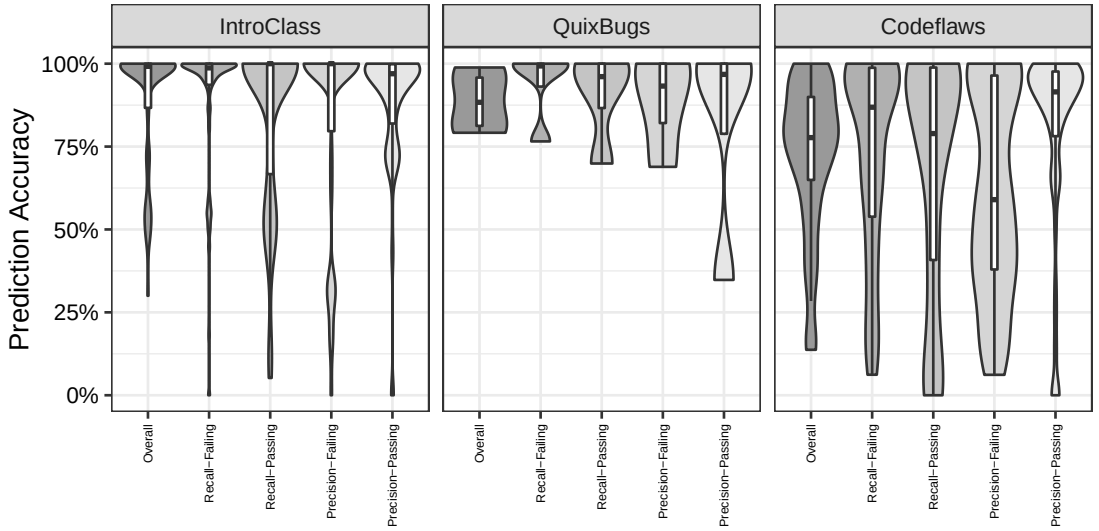
For each subject, we computed the average of these metrics over the 30 runs.

> For the majority of subjects, the automatic oracles accurately predict the labels of more than 92% of the manually labelled test inputs. Even though GRAMMAR2FIX has seen only one failing input, the automatic oracles correctly identify more than 97% of the failing tests in most subjects. The precision and recall for the passing inputs are above 95% for the median subject.
> In each benchmark, the median overall accuracy is $> 75\%$, and the median conditional accuracy-failing (recall failing) is $> 85\%$.

According to the results, the automatic oracles learnt by GRAMMAR2FIX classify test inputs with significantly high accuracy in most subjects. This implies that this grammar inference approach can accurately induce a grammar to explain the structure of the failing inputs of a given buggy program. The high conditional accuracy for passing and



(a) Overall distribution



(b) Benchmark-wise distribution

FIGURE 5.9: Violin plots of the distributions in overall accuracy, precision and recall. Figure 5.9(a) Overall distribution over the three benchmarks and Figure 5.9(b) distributions under each benchmark

failing inputs indicates that the induced grammar is not only effective in recognising failing inputs, but can also accurately reject passing inputs.

Result. GRAMMAR2FIX *induces high-quality grammars that explain the failure conditions of many real-world bugs in string processing programs. These grammars can be used as automatic test oracles*

5.6.2 Contributions From the Heuristics

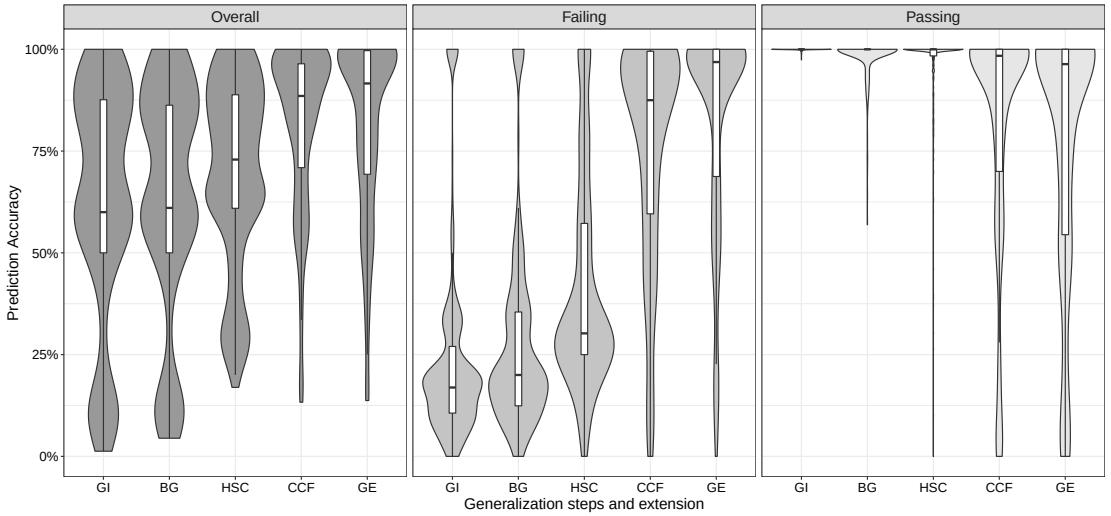


FIGURE 5.10: Violin plots of the prediction accuracy distributions after each step of GRAMMAR2FIX across all subjects. The steps are grammar inference (GI), basic level generalization (BG), handling special cases (HSC), character class finding (CCF), and grammar extension (GE).

Figure 5.10 shows the distributions in overall accuracy, conditional accuracy-failing and conditional-passing after each step of GRAMMAR2FIX. Again, for each subject, we computed the average values over 30 runs of the different steps of GRAMMAR2FIX.

> The median overall accuracy increases from 60% for the first step (GI) to 92% for the last step (GE). The median conditional accuracy-failing (recall failing) increases from 17% for the first step (GI) to 97% for the last step (GE).

The median conditional accuracy-failing (recall failing) gradually increases from GI to HSC. There is a significant increase from HSC to CCF in the median of this metric (Figure 5.10 - middle). In addition, there are slight decreases in the median conditional accuracy - passing (recall passing) in CCF and GE (Figure 5.10 - right). The median overall accuracy gradually increases with the heuristics.

Result. *The different heuristics used in* GRAMMAR2FIX *positively affect the oracle quality. These heuristics significantly improve the ability of the automatic oracles to identify failing tests.*

5.6.3 Human Labelling Effort

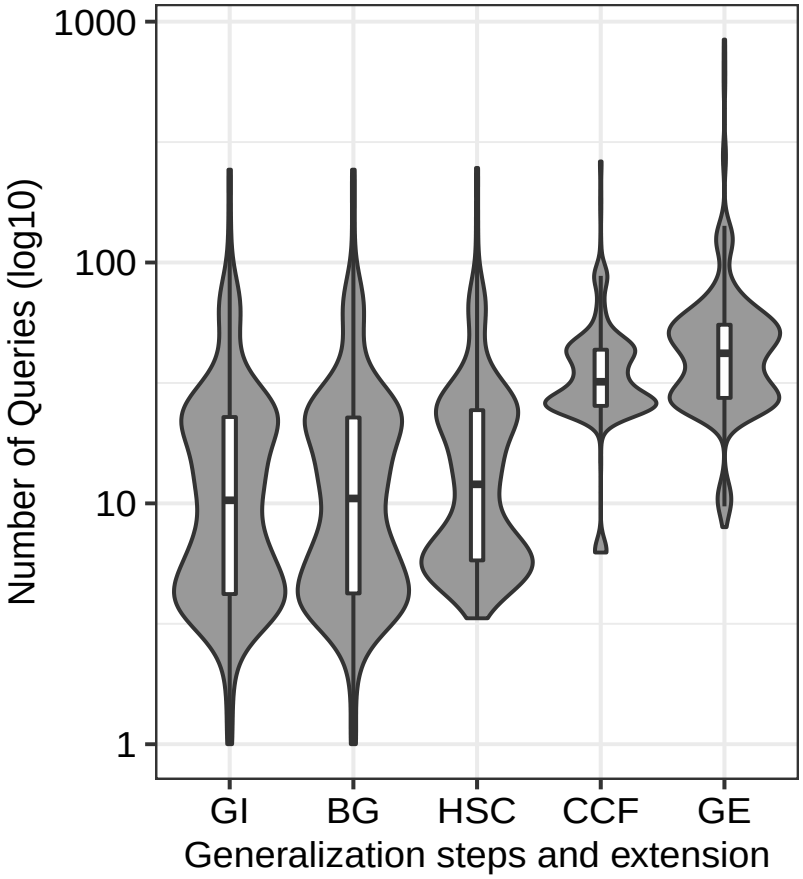


FIGURE 5.11: Violin plots (log-scale) of the cumulative number of queries to the human after each step of GRAMMAR2FIX as a distribution across all subjects.

GRAMMAR2FIX Step	Human labelling queries	
	Mean	Median
GI	17.18	10.30
BG	17.25	10.48
HSC	19.11	12
CCF	37.95	31.98
GE	52.52	42

TABLE 5.5: Mean and median of the cumulative number of queries to the human after each step of GRAMMAR2FIX

Figure 5.11 shows the labelling effort in terms of the number of queries to the human after each step. The violin-plots are in log-scale. Table 5.5 shows the mean and median of the number of human queries in each step.

> The median number of human queries increases with the generalization and extension steps of GRAMMAR2FIX. Nevertheless, the number of human queries with all the steps is 42 for the median subject.

From GI to BG and BG to HSC, there are smaller increases in the median human labelling queries. The highest increase is from HSC to CCF. The fixed substitution iterations (Section 5.4.3.3) in CCF is the reason for this increase.

We find that the median number of queries sent to the human is reasonable. Firstly, given a string input and the corresponding program output, the human would only need to provide a "Yes" or "No" to the question: "Does this program process this string correctly?". Secondly, these queries can be distributed among multiple people reporting the same bug.

Result. *For the majority of bugs,* GRAMMAR2FIX *infers grammar using a reasonable number of human queries.*

5.6.4 Discussion

Given a failing input, the grammar inference process of GRAMMAR2FIX explores the root cause of the failure and how it is related to the failing inputs of the bug. The root cause is the minimal failing input (f_{min}) given by *ddmin*. GRAMMAR2FIX can accurately model different relationships between f_{min} and the other failing inputs, thus leading to high-quality grammar oracles for many subjects. However, we found that GRAMMAR2FIX induces less accurate grammar in a few subjects. We identified two main reasons for having this. Firstly, GRAMMAR2FIX does not model dependencies between complementary self-transitions (Section 5.4.3.1), as finding such dependencies requires more human-labelled test inputs. However, such dependencies could be necessary to accurately describe the failing input patterns of some bugs. Secondly, in some subjects, the mutational fuzzing iteration in GE (Section 5.4.4) could be insufficient to identify all the constituents of the structure of failing inputs. This issue is particularly evident

in the programs that accept structured string inputs. To deal with this kind of program, structure-aware fuzzing might be helpful (e.g., AFLSmart [195]).

According to the results in Section 5.6.2, the heuristics used in GRAMMAR2FIX improve the oracle quality. The CCF step significantly improves the ability of identifying failing inputs (Figure 5.10). The reason is that the DFA is expanded to a collection of DFAs by exploring more minimal failing inputs, which significantly avoids overfitting the grammar to the training examples. However, we observe a slight decrease in conditional accuracy-passing from HSC to CCF. The key reason is that the bugs having passing inputs of the same length of f_{min}. In such a bug, if Algorithm 7 does not find such passing inputs within the limited number of substitution iterations, Case 2 in Algorithm 7 concludes that the unique characters of f_{min} can be substituted with any set of unique characters. The result is that the collection of DFAs accepts some passing inputs; i.e., the oracle incorrectly classifies some passing tests as failing.

We observed that GRAMMAR2FIX infers high-quality grammar for failing inputs, using a reasonable number of human queries. Unlike LEARN2FIX, GRAMMAR2FIX cannot be used under a limited number of human queries. The reason is that *ddmin* [77], the input minimization algorithm, follows a deterministic number of steps. In each step, the generated test input should be labelled to proceed with the algorithm. As GRAMMAR2FIX cannot proceed without finding the minimal failing input (f_{min}), we cannot set a maximum limit for the human queries. Therefore, more queries are required to minimize a significantly longer failing input. This is why few subjects need a large number of human queries in the grammar inference process.

In this work, we did not test GRAMMAR2FIX with the adversarial learning techniques (Section 4.7) and culprit constraint-based approach (Section 4.8) proposed in Chapter 4. The key reason is that the objective of the test generation methods in GRAMMAR2FIX is refining the grammar rather than exploring more failing tests. Thus, the two additional approaches proposed for LEARN2FIX are not compatible with the test generation methods of GRAMMAR2FIX. Nevertheless, the GE step can be seen as an adversarial approach, as it refines the current grammar by exploring the failing tests that the current grammar cannot identify.

5.7 Extending GRAMMAR2FIX for Other Structured Inputs

GRAMMAR2FIX can be extended for other structured inputs used in computer programs. Firstly, the *alphabet*, i.e., the terminals [137], of the targeted inputs should be identified. *ddmin* (Section 5.4.1) should be preformed according to the alphabet. Also, a *complementary self-transition*, introduced in BG (Section 5.4.3.1), should happen under any character in the alphabet that does not involve in the outgoing inter-state transitions of the state. Secondly, HSC, CCF and GE steps should use the characters of the alphabet to generate valid structured inputs.

In some programs taking structured inputs, a set of syntactic rules is defined on the alphabet, i.e., an input grammar, to determine the validity of the inputs (e.g. compilers). When using GRAMMAR2FIX with this kind of input, the invalid inputs generated in the grammar inference can be neglected. The *program under test* (PUT) itself can be used to exclude invalid inputs. The reason is that this kind of program is usually embedded with the rules to determine the valid inputs; therefore, it can reject invalid inputs. For example, compilers reject programs containing syntactically incorrect statements.

The output of GRAMMAR2FIX is a *regular grammar*, represented by one or more DFAs. To describe the failure conditions associated with highly structured inputs, more complex grammars (e.g. context free grammar, context sensitive grammar etc. [196]) might be required. Accordingly, the grammar inference (Section 5.4.2) and grammar generalization(Section 5.4.3) might require some refinements.

5.8 Conclusions

The main conclusions of the experiments regarding GRAMMAR2FIX can be summarized as follows.

1. *Oracle Quality*

    - GRAMMAR2FIX produces high-quality automatic test oracles as grammars describing the pattern of failing inputs for string processing programs.

2. *Human Labelling Effort*

    - GRAMMAR2FIX uses a reasonable number of queries to the human in most subjects to infer grammar.

3. *Contributions of the heuristics*

    - The different heuristics of GRAMMAR2FIX improve the accuracy of the grammar describing the structure of failing inputs.

GRAMMAR2FIX is capable of generating high-quality automatic oracles (grammar oracles) for semantic bugs in string processing programs. A grammar oracle accurately describes the structure of the failing inputs of the semantic bug. Similar to LEARN2FIX, GRAMMAR2FIX oracle learning systematically interacts with the human. This oracle learning technique can be extended as an oracle learning framework for structured inputs. Therefore, GRAMMAR2FIX provides an answer to **RQ.2**. Also, this is a contribution under **C.2**.

Chapter 6

Oracle Learning to Guide Automated Program Repair

This chapter demonstrates how LEARN2FIX and GRAMMAR2FIX can be applied to guide *Automated Program Repair* (APR) to fix semantic bugs, with a focus on **RQ.3**. To summarize, we convert the test suite generated in oracle learning to a repair test suite for the bug. Then, we use this repair test suite with APR. This chapter presents the experimental analysis of our approach.

6.1 Motivation

Automated Program Repair(APR) [7, 9] reduces the burden of manual bug fixing in rapidly evolving software systems. *Test-driven* APR techniques (e.g. *GenProg* [102], *Angelix* [107], etc.) can be applied to repair different types of programs [7]. *Repair overfitting* [112], i.e., a lack of generalizability of auto-generated patches, is a critical problem in test-driven automated program repair. An overfitting patch might not completely fix the bug and may actually introduce more faults to the system. One method to resolve the problem of repair overfitting is to improve the *repair test suite* (Section 2.5.2).

A *repair suite* contains some *passing* and *failing* tests. The failing tests exercise the faulty behaviour to be fixed, while the passing tests indicate the behaviour that should not be changed. A *test-driven* APR technique uses the repair test suite to identify the code locations that are likely to be faulty (i.e., fault localization [75]) in the given

program. Using fault localization information, the APR technique changes the buggy program so that it passes all the test cases in the repair test suite. Thus, the repair test suite determines the quality of the auto-generated patch. If the repair test suite contains enough failing tests exercising the bug and passing test cases indicating the behaviours that should not be changed, the APR technique can produce a non-overfitting patch for the bug. We study how to use our oracle learning techniques to produce such repair test suites to guide APR to generate high-quality patches for semantic bugs.

Both LEARN2FIX (Chapter 4) and GRAMMAR2FIX (Chapter 5) generate a test suite in learning an oracle for a semantic bug. The results in Section 4.6.2 show that LEARN2FIX generates test suites containing more failing tests of semantic bugs in oracle learning. In LEARN2FIX, we used the DECIDE2LABEL-algorithm (Algorithm 2) and *mutational fuzzing* [197] to maximize the human labelling of failing tests. Similarly, the probability of generating failing tests is higher in GRAMMAR2FIX, as it uses *delta debugging minimization (ddmin)* [77] and some systematic test generation approaches. Therefore, in this chapter, we experimentally analyse the applicability of test suites generated in oracle learning as repair test suites with APR.

6.2 Methodology

Given a buggy program, we first allow the learning technique (LEARN2FIX or GRAMMAR2FIX) to generate an automatic oracle and a test suite. Next, we use the test suite to create a repair test suite for the bug.

6.2.1 Creating a Repair Test Suite

A test suite returned by LEARN2FIX or GRAMMAR2FIX contains some failing inputs and zero or more passing inputs. Also, it contains the *program output* of each test input. As the program under test (PUT) produces incorrect outputs for failing inputs, the *program output* of a failing test case is not the expected, correct output.

In a repair test suite used in test-driven APR, each test case should consist of an input and its expected, correct output. Thus, we replace *the output of each failing test case with its expected, correct output* to convert a test suite obtained in oracle learning to a

repair test suite. There is no need to change the output of passing test cases, as the program produces correct, expected outputs for those (Algorithm 9). We name these repair test suites *auto-generated* repair test suites in the experiments.

Algorithm 9 Create Repair Test Suite

Input: T : Test suite given by oracle learning

```
for t ∈ T do
    if t is failing then
        Replace the output of t with its expected, correct output
    end if
end for
```

6.2.2 Test-Driven Automated Program Repair Techniques

A test-driven automated program repair technique uses a repair test suite and the buggy program to generate a fix for the bug. To evaluate our approach, we use the following test-driven APR techniques.

i. GenProg [102]

ii. Angelix [107]

GenProg [102] is a *heuristic repair* technique that has been developed to repair C programs. As the first step, it performs *spectrum-based fault localization* [7] based on the repair test suite and assigns a weight to each statement of the program. The statements executed by the failing tests receive higher weights than other statements. If a statement is executed only by the passing tests, it receives a zero weight. Next, using the *abstract syntax tree* (AST) of the program, *GenProg* produces program variants by *genetic programming* [198]. In this process, mutation operators are applied to highly weighted program statements. The statements with zero weights are not modified. *GenProg* continues modifying the program until a variant passing all the tests in the repair test suite is found.

Angelix [107] is a *constraint-based repair* technique that has been developed to repair C programs. Firstly, it performs a semantics-preserving program transformation. Next, *Angelix* performs fault localization using the Jaccard formula [101] using the repair test suite to identify suspicious program locations. Constructing the *repair constraint* is the

most important task in *constraint-based repair techniques* [9]. *Angelix* develops the *repair constraint* as an *angelic forest* [199]. This process uses the information obtained during the fault localization. Finally, *Angelix* employs *component-based repair synthesis*(CBRS) to generate a patch for the bug. CBRS uses the *angelic forest* created previously as a specification in this process.

We use the test-driven APR techniques listed above due to their capability to repair large programs cost-effectively. To use our auto-generated test suites with these APR techniques, we extend the workflows of LEARN2FIX and GRAMMAR2FIX as shown in Figure 6.1 and Figure 6.2, respectively.

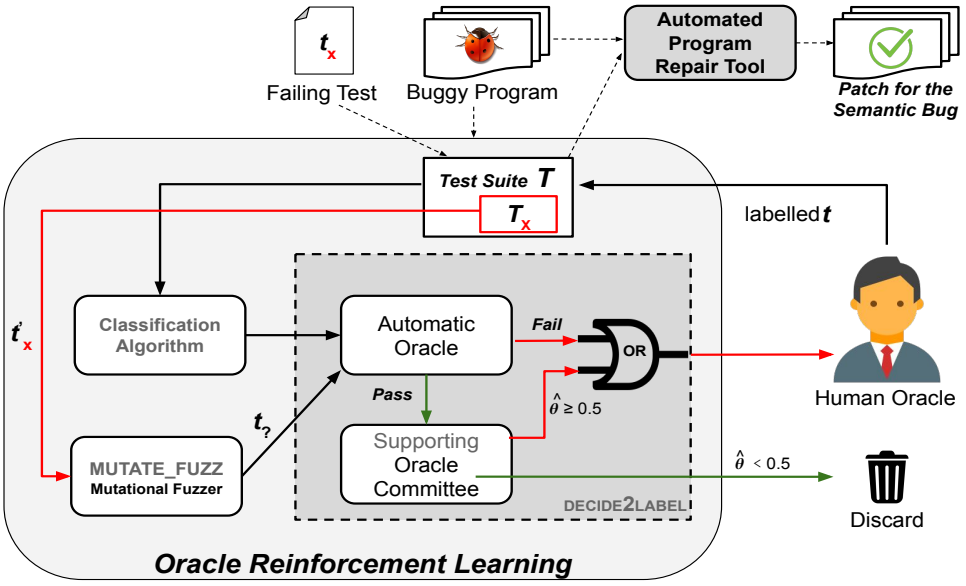


FIGURE 6.1: LEARN2FIX workflow with automated program repair

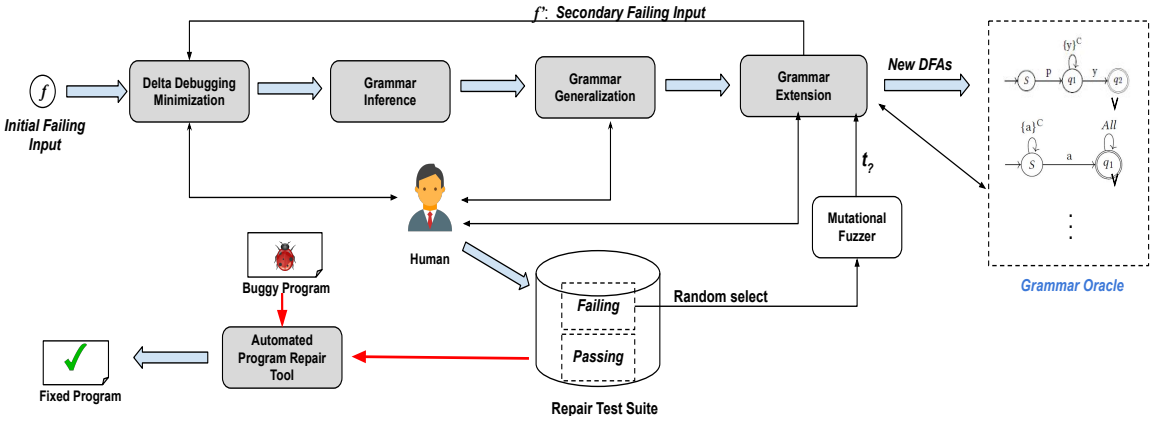


FIGURE 6.2: GRAMMAR2FIX workflow with automated program repair

6.3 Experimental Setup and Evaluation

For these experiments, under each learning technique (LEARN2FIX and GRAMMAR2FIX), we selected the same set of subjects from *Codeflaws* [177] that was used in the oracle learning experiments (Section 4.5.1 and 5.5.1). There were 552 subjects for LEARN2FIX and 152 for GRAMMAR2FIX. We selected subjects from *Codeflaws* for the following reasons.

i. *Codeflaws* has been setup to work with *GenProg* and *Angelix*

ii. *Codeflaws* provides a separate repair validation test suite, i.e., heldout test suite for each subject.

Codeflaws provides a *manually* created repair test suite for each subject. In the experiments, we compared our *auto-generated* repair test suites with the *manual* repair test suites, under both *GenProg* and *Angelix*.

For each subject, after the completion of oracle learning, we created the *auto-generated* repair test suite as described in Section 6.2.1. Then, we separately used the *manual* repair test suite (given by *Codeflaws*) and the *auto-generated* repair test suite with the APR tools in Section 6.2.2. We allocated 10 minutes to each APR tool to generate a patch under each repair test suite. If a patch was generated, we counted the number of tests in the heldout test suite passed on the patched program.

The performance of the manual and auto-generated repair test suites was measured in terms of the following metrics.

i. Repairability: Proportion of subjects successfully repaired (Equation 6.1).

ii. Validation Score: Proportion of validation tests passed on the patched program (Equation 6.2).

$$Repairability = \frac{Number\ of\ subjects\ successfully\ repaired}{Total\ number\ of\ subjects} \tag{6.1}$$

$$Validation\ Score = \frac{\begin{array}{c}Number\ of\ validation\ tests\ passed\ on\ the\\ patched\ program\end{array}}{\begin{array}{c}Total\ number\ of\ tests\ in\ the\\ validation\ test\ suite\end{array}} \tag{6.2}$$

In LEARN2FIX, we repeated this repair experiment under each classification algorithm listed in Section 4.4.2.

To mitigate the impact of randomness and to gain statistical power for the experimental results, we repeated each experiment 30 times for each subject.

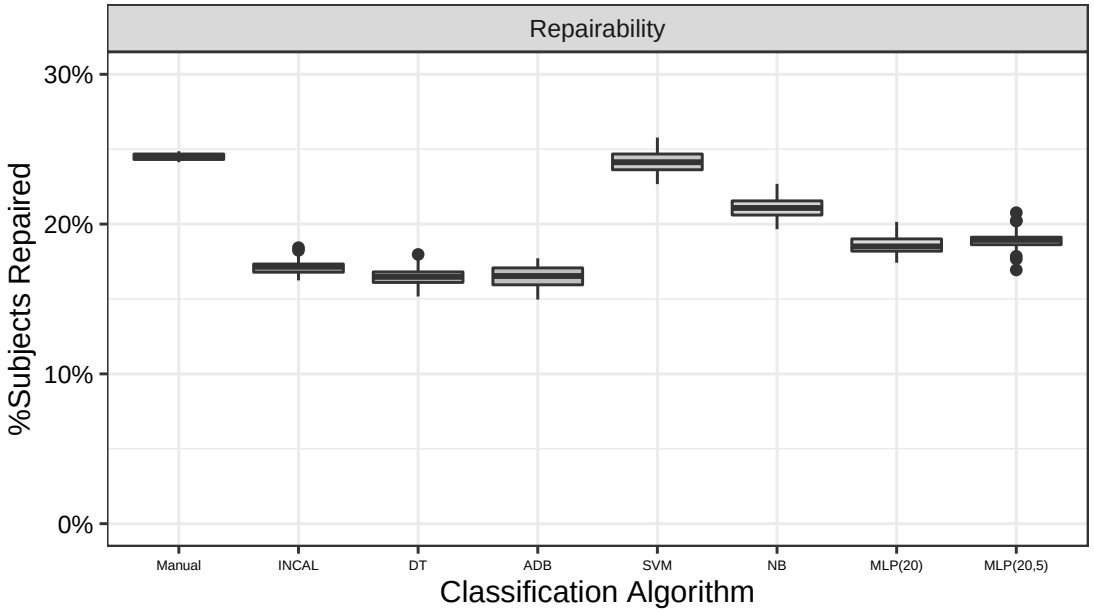
6.4 Experimental Results and Discussion

6.4.1 LEARN2FIX

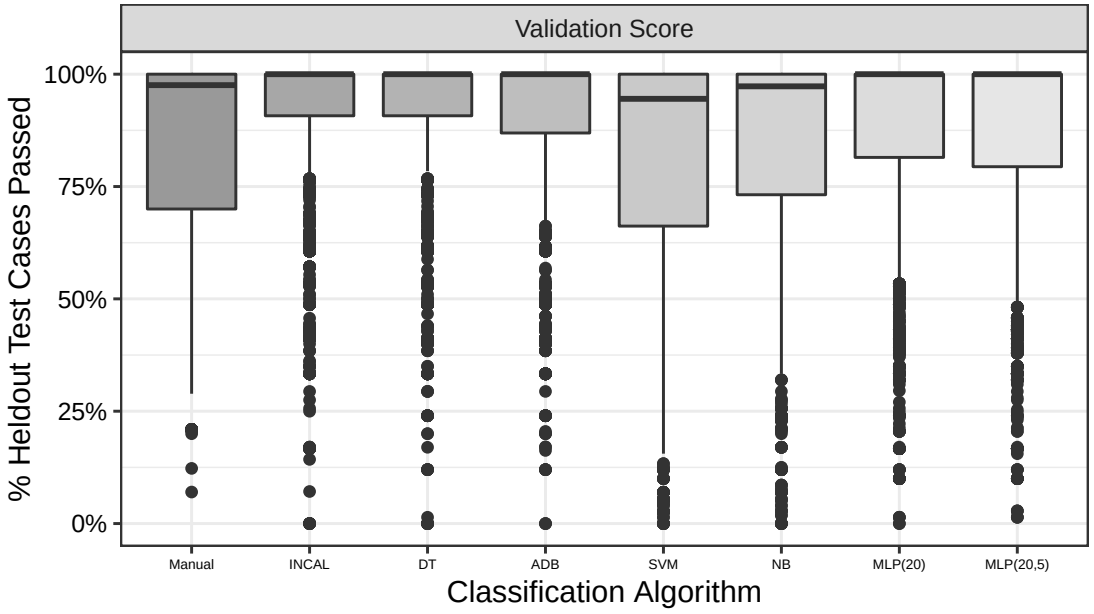
Test Suite	GenProg				Angelix			
	Repairability (%)		Validation Score(%)		Repairability (%)		Validation Score(%)	
	Mean	Median	Mean	Median	Mean	Median	Mean	Median
Auto-generated Interpolation-based								
INCAL	17.14	17.15	90.35	100	15.66	15.57	90.65	100
Decision Tree	16.48	16.48	90.45	100	16.49	16.45	90.66	100
Ada Boost	16.47	16.54	89.71	100	16.19	16.29	90.44	100
Auto-generated Approximation-based								
SVM	24.15	24.13	81.77	94.52	22.71	22.70	82.64	91.67
Naïve Bayes	21.06	21.08	84.64	97.3	20.97	20.89	84.21	94.44
MLP (20)	18.64	18.51	86.75	100	18.49	18.50	88.51	97.14
MLP (20,5)	18.91	18.94	86.63	100	18.41	18.60	88.13	95.83
Manual								
Manual	23.52	24.50	85.14	97.56	25.53	25.50	83.94	91.67

TABLE 6.1: Mean and median values of the repairability and validation scores obtained under GenProg and Angelix

Figure 6.3 and Figure 6.4 show the results obtained under *GenProg* and *Angelix*, respectively. Table 6.1 summarizes the results of the program repair experiments, presenting the mean and median values of the metrics.

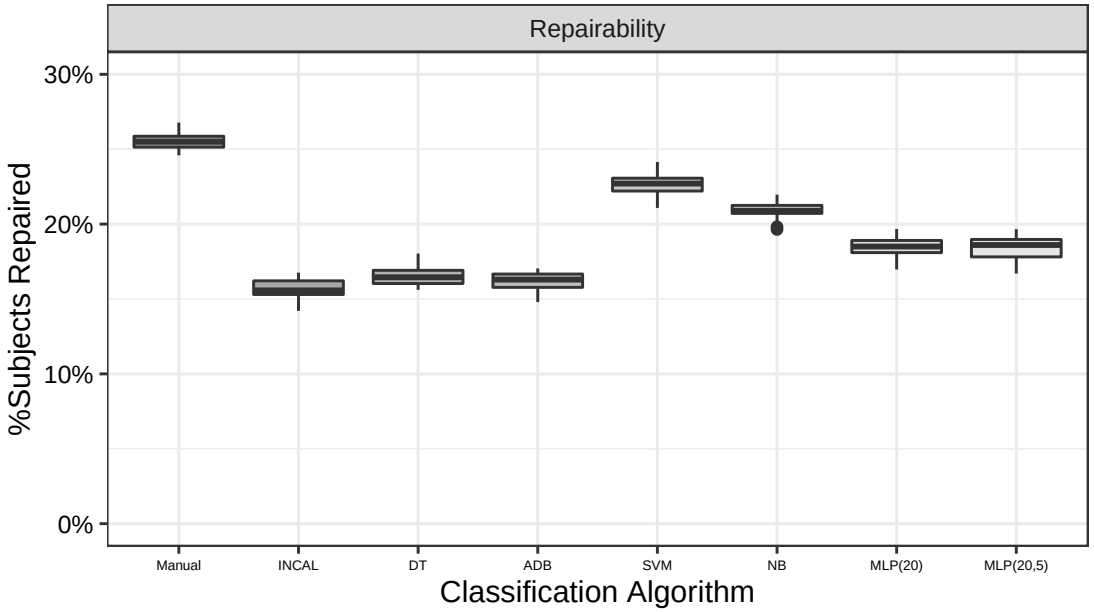


(a) Repairability

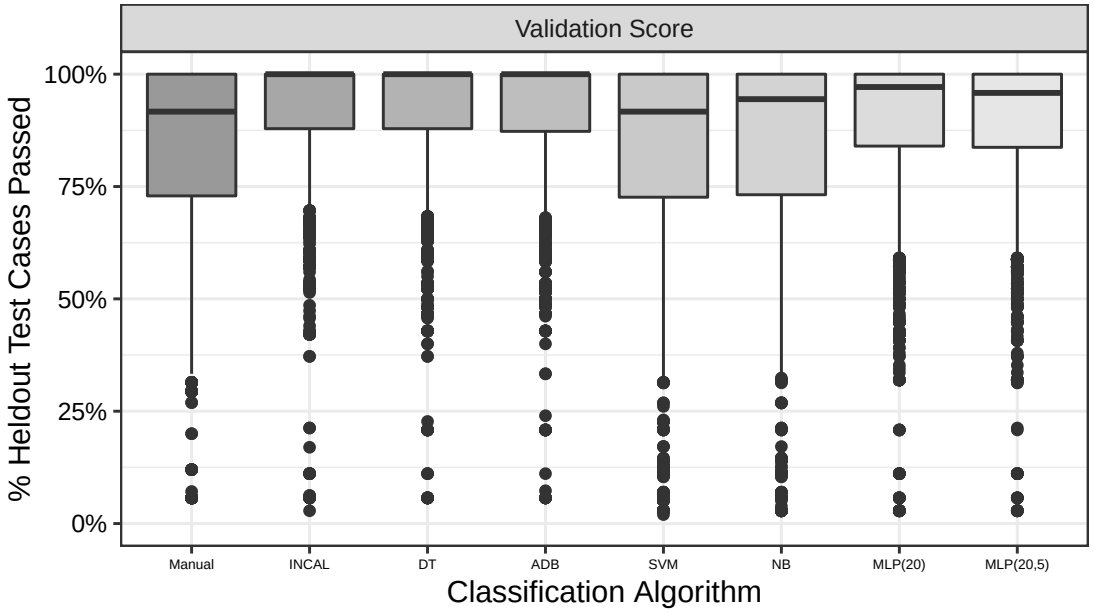


(b) Validation Score

FIGURE 6.3: Distributions of repairability and validation scores of LEARN2FIX obtained under GenProg



(a) Repairability



(b) Validation Score

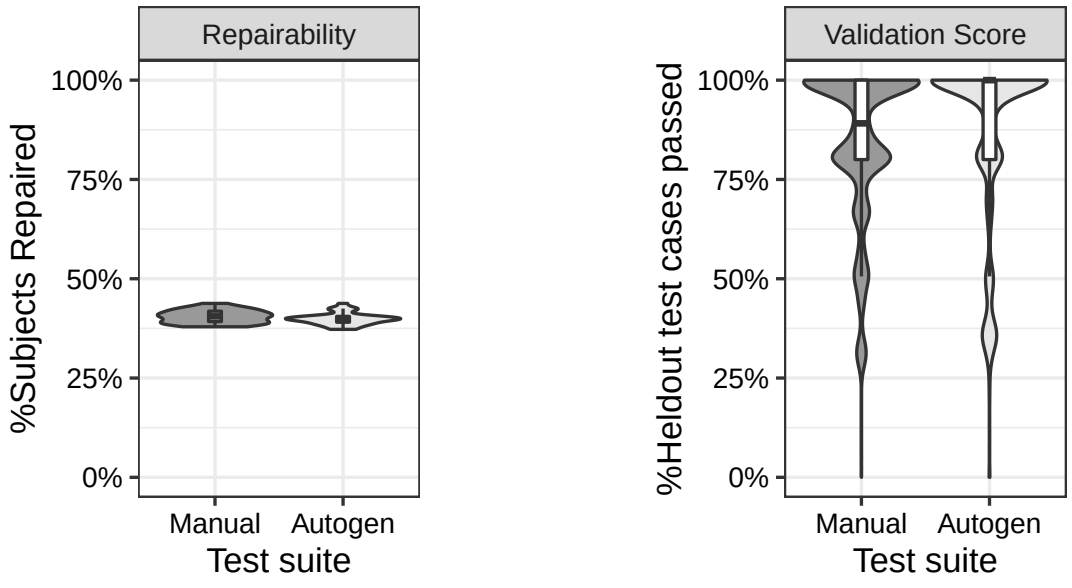
FIGURE 6.4: Distributions of repairability and validation scores of LEARN2FIX obtained under Angelix

In both APR techniques, the auto-generated test suites under most classification algorithms (except SVM and Naïve Bayes) outperform the manual test suites in terms of the validation score of the generated patches. The median validation score is 100% in all the interpolation-based classifiers in both APR techniques. Both types of test suites can repair less than 30% of the selected subjects. Also, the manual test suites can repair more subjects than the auto-generated test suites by most classification algorithms.

The interpolation-based classification algorithms performed better with LEARN2FIX (Section 4.6.3), and those produce repair test suites in the oracle learning that lead to high-quality patches in both *GenProg* and *Angelix*. Although the manual test suites can repair more subjects than the auto-generated test suites of most classification algorithms, the patches are not accurate as in the auto-generated repair test suites.

Result. *Auto-generated repair test suites by* LEARN2FIX *with interpolation-based classification algorithms produce high-quality patches with different automated program repair techniques.*

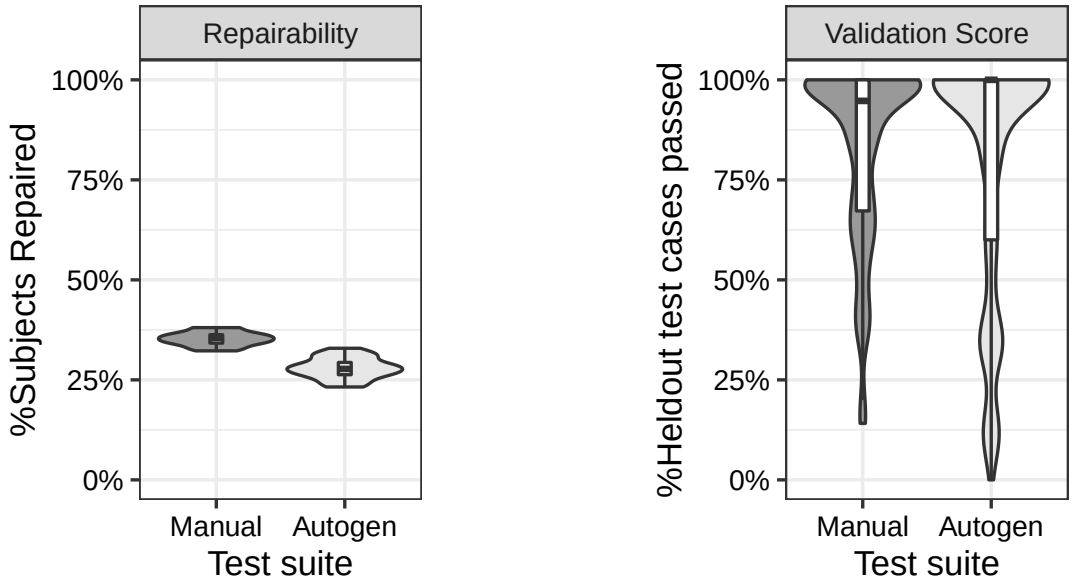
6.4.2 GRAMMAR2FIX



(a) Repairability

(b) Validation Score

FIGURE 6.5: Distributions of repairability and validation scores of GRAMMAR2FIX obtained under GenProg



(a) Repairability

(b) Validation Score

FIGURE 6.6: Distributions of repairability and validation Scores of GRAMMAR2FIX obtained under Angelix

Test Suite	GenProg				Angelix			
	Repairability (%)		Validation Score(%)		Repairability (%)		Validation Score(%)	
	Mean	Median	Mean	Median	Mean	Median	Mean	Median
Auto-generated	39.89	39.87	85.37	100	27.94	27.74	78.10	100
Manual	40.46	40.52	84.09	89.12	35.26	35.48	82.24	94.74

TABLE 6.2: Mean and median values of the repairability and validation scores obtained under GenProg and Angelix

Figure 6.5 and Figure 6.6 show the results under *GenProg* and *Angelix*, respectively. Table 6.2 summarizes the results of program repair experiments, indicating the mean and median values of the metrics.

In most repairable subjects, both APR techniques generate patches that pass 100% of the held-out test suite with the auto-generated test suite by GRAMMAR2FIX. The manual test suites do not lead to producing patches with this much accuracy. However, the manual test suites can repair more subjects than the auto-generated test suites.

Result. *Auto-generated repair test suites generated by* GRAMMAR2FIX *generate high-quality patches with different automated program techniques*

6.4.3 Discussion

The experimental results (Section 6.4.1 and Section 6.4.2) suggest that the *auto-generated* repair test suites in LEARN2FIX and GRAMMAR2FIX oracle learning can be used as repair test suites for semantic bugs with APR. With both APR techniques, the auto-generated test suites lead to producing high-quality patches than the manual test suites given by the benchmark. Although the APR techniques can repair more subjects with the manual repair test suites than the auto-generated repair test suites, the patches are not as accurate as in the auto-generated test suites (lower validation score). This implies that the manual test suites lead to repair overfitting. Therefore, we conclude that auto-generated repair test suites are better in program repair, as the production of less accurate patches is detrimental. Also, this repair test suite generation method can be generalized over different test-driven APR techniques.

The key reason for the higher patch quality is that the auto-generated repair test suites contain more failing tests than the manual test suites. For the majority of subjects, the failing test cases of an auto-generated test suite can exercise all the behaviours of the bug. We have identified that, in most subjects, the manual test suite contains only one failing input of the bug. Thus, the manual test suites lead to overfitting or incorrect patches with the APR techniques. For this reason, the patches produced with the manual test suites pass fewer tests in the heldout test suites than the ones generated with the auto-generated test suites.

In LEARN2FIX, the median validation score is 100% in both APR techniques with the interpolation-based classification algorithms. Under these algorithms, the *auto-generated* repair test suites outperform the *manual* repair test suites in terms of validation score. The interpolation-based classification algorithms produce high quality oracles for most subjects (Section 4.6.3). Thus, we conclude that classification algorithms that produce high-quality automatic oracles generate better repair test suites with LEARN2FIX for *Angelix* and *GenProg*. LEARN2FIX sends fewer failing tests to the human when using the SVM classification algorithm (Table 4.3). For this reason, the auto-generated repair test suites under SVM contain fewer failing tests compared with the other classification algorithms. Therefore, the patches generated with the auto-generated repair test suites become less accurate. This is the reason why the auto-generated repair test suites using SVM do not outperform the manual repair test suites in both APR techniques.

In GRAMMAR2FIX, the median validation score under both APR techniques is 100% (Table 6.2). The *auto-generated* repair test suites under GRAMMAR2FIX outperform *manual* repair test suites in terms of validation score. This outcome implies that *delta debugging minimization (ddmin)* [77] and the other systematic test generation methods used in GRAMMAR2FIX are able to develop high-quality repair test suites.

In LEARN2FIX, less than 30% of the experimental subjects that can be repaired with the manual and auto-generated test suites using APR techniques. In GRAMMAR2FIX, this percentage is less than 41%. This is due to the problems associated with APR techniques. We have observed that when the repair test suite contains more failing tests, the APR tool is unable to alter the program to make all the failing tests pass. As described before, our auto-generated test suites contain more failing test cases, hence the reduction in repairability. In some subjects, the APR tools exceed the time allocated to generate repair, which reduces the repairability of both types of test suites.

Manually finding a repair test suite to produce an accurate repair for a semantic bug is challenging. Our oracle learning techniques address this problem by providing an automated means to generate and select test cases for human labelling. The human can easily be involved in the process of creating a repair test suite for the bug. Moreover, our auto-generated test suites produce high-quality repairs with two test-driven automated program repair approaches (GenProg: *heuristic-based*, Angelix: *constraint-based*). Thus, in terms of repair test suite generation, our oracle learning techniques can be generalized over different test-driven APR techniques.

6.5 Conclusion

Our oracle learning techniques can be used to generate repair test suites that lead to high-quality repairs for many real-world semantic bugs. Also, these learning techniques facilitate *human-in-the-loop interactive program repair*. Figure 6.7 depicts this framework. In this manner, our oracle learning process can be exploited to guide APR to fix semantic bugs. Therefore, the repair test suite generation method introduced in this chapter answers **RQ.3**. Also, it is a contribution under **C.3**

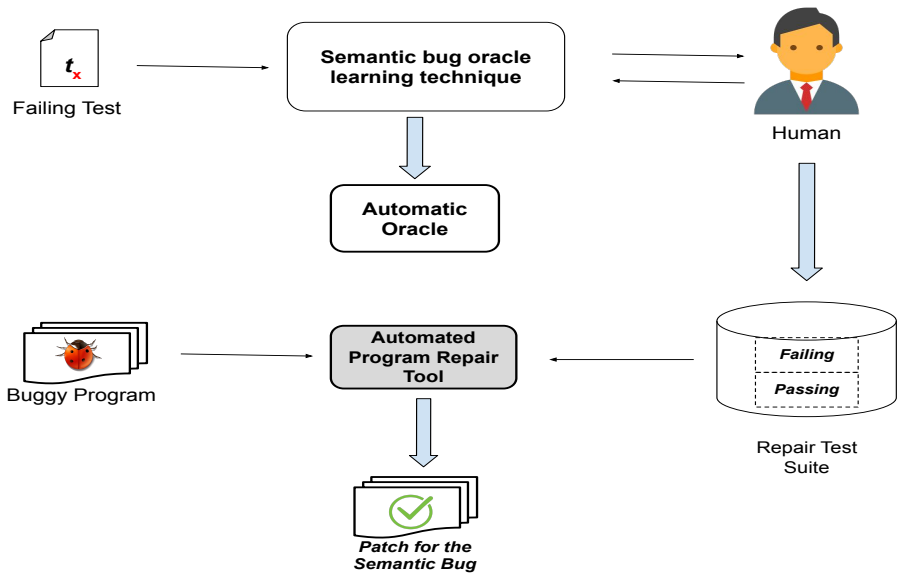


FIGURE 6.7: LEARN2FIX workflow with automated program repair

Chapter 7

Overall Conclusions

Test oracle automation is an important task to facilitate automated software testing. However, several studies suggest that this topic has received less research attention. In addition, developing automatic test oracles for semantic bugs is challenging, as those are *application specific*. The most practical and reliable source to identify the test failures of a semantic bug is the human (the user or the developer). Hence, it is necessary to interact with the human to derive an automatic oracle for a semantic bug. However, the existing test oracle automation techniques have not focused on the problem of systematically learning an automatic oracle from a human. This thesis mainly presents a solution to this problem through two *human-in-the-loop oracle learning techniques*. Moreover, it presents a *semi-automatic* approach to guide automated program repair to fix a semantic bug. These solutions make significant contributions to test oracle automation and automated program repair.

We identified some machine learning-based for test oracle automation. Most of these works focus on modelling the expected behaviour of the system under test and using it to predict the output of new test cases. However, modelling the expected behaviour of a software application can be a difficult task. In contrast, learning the condition under which a bug is exposed, i.e., the failure condition of a bug, is an easier task. Also, the failure condition of a bug is independent of program size. An automatic oracle trained by our techniques expresses a failure condition in terms of program inputs and outputs. This is a black-box representation, and such a representation can be easily used in debugging and program repair.

Both LEARN2FIX and GRAMMAR2FIX learn automatic oracles, addressing many practical problems associated with semantic bugs. One such problem is that a semantic bug is usually reported with a single input exposing it. Indeed, a single failing input is insufficient for learning about the bug. Our learning techniques have their own test generation methods that guide oracle learning. LEARN2FIX selects and sends test cases that are likely to be failing to the human, while GRAMMAR2FIX generates test inputs according to a systematic order. These two strategies resolve the problem of obtaining training data for oracle learning. Moreover, our learning techniques use some techniques to systematically interact with the human. The experimental results indicate that these techniques enable the learning of accurate oracles for many real-world semantic bugs with a reasonable number of human labelling queries.

In both our learning techniques, the human is asked an "Yes/No" question of the form "For the input $\vec{i}$, the program produces the output o; is the bug observed?" to obtain the label of a test case. Usually, a user or a developer who participates to debug a program knows its expected, correct behaviour. Such a person can easily answer this kind of labelling queries. As our learning techniques can train automatic test oracles with a reasonable number of human labelling queries, those can be used with actual human participants.

An automatic oracle trained by our learning techniques describes the condition under which a semantic bug is exposed (Section 1.2). Hence, the automatic oracle can be used to develop a specification for the semantic bug. Such bug specifications are useful for software developers in many ways. However, our oracle learning techniques might require some improvements to achieve this objective. For instance, the constraints or grammars in automatic oracles should be converted to more human-readable forms for developing bug specifications.

In LEARN2FIX (Chapter 4), we paid significant attention to handling the *class imbalance problem* [17] in oracle learning. By modifying the work of Holub et al. [166], we developed a method for selecting test cases that are likely to be failing to address this problem (DECIDE2LABEL-Algorithm 2). The class imbalance problem arises when machine learning is applied to many real-world applications. The reason is the difficulty in finding a balanced training dataset. The DECIDE2LABEL-algorithm can be applied to such situations. It can even be customised for multi-class scenarios.

We identified that oracle learning with string inputs is more difficult than with numeric inputs. The reason is the complexity of string inputs. GRAMMAR2FIX effectively copes with this complexity through its *zooming in and zooming out* strategy. First, it explores the minimal cause of the failure. Then, identifying the relationships between the minimal cause and the other failing string inputs, GRAMMAR2FIX develops a grammar for all the failing inputs. This approach can be generalized for general regular grammar inference [142] in machine learning when an oracle exists to answer membership queries [123].

In both learning techniques, the auto-generated test cases in oracle learning create a high-quality repair test suite [9] for the bug. We observed that these repair test suites led to high-quality repairs with both *GenProg* [102] and *Angelix* [107]. The reason for this outcome is that an auto-generated test suite by our learning techniques contains more failing tests of the bug. The test generation methods and active learning techniques used in our learning techniques help to collect more failing tests of a bug. Finding this type of repair test suite manually, especially for a semantic bug, would be extremely difficult. This capability of our learning techniques addresses the repair overfitting problem in automated program repair (APR).

The ability of auto-generated test suites to generate high-quality patches implies a few further aspects. Firstly, both *GenProg* [102] and *Angelix* [107] can accurately identify faulty code segments, i.e., fault localization, with our auto-generated test suites. These APR techniques use spectrum-based fault localization (SBFL) techniques [80]. In case of enough failing tests, SBFL is successful with our auto-generated test suites. Secondly, the auto-generated test suites help *Angelix* to synthesise accurate *repair constraints*. In *constraint-based repair* [9] techniques such as *Angelix*, the repair constraint explains the characteristics of the patch to be generated. Having more failing tests in the repair test suite improves the accuracy of the repair constraint, thus improving the quality of the patch.

In relation to program repair, we observed that our auto-generated test suites could repair fewer subjects compared to the manual test suites given by the benchmark. This is because of the issues associated with APR tools. When the test suite has more failing tests, the APR tool might not be able to generate a patch passing all the tests. Therefore,

test driven APR techniques require some improvements to generate high quality patches for bugs.

Detecting and fixing semantic bugs is a challenging task in software testing. The application-specific nature of semantic bugs is the key reason for this difficulty. Our oracle learning techniques provide a *semi-automatic* solution to this problem. Once a semantic bug is reported with a single input exposing it, our oracle learning techniques can generate a high-quality repair test suite by systematically interacting with the human (the user or developer). This is an interactive program repair environment that even a person without programming knowledge can participate in. Also, this environment can be used with different automated program repair techniques. Therefore, LEARN2FIX and GRAMMAR2FIX facilitate *human-in-the-loop interactive program repair*.

Chapter 8

Future Work

We have identified some limitations in our two oracle learning techniques, LEARN2FIX (Chapter 4) and GRAMMAR2FIX (Chapter 5), and possible research areas that can be improved by our findings. Based on those, we propose the following research directions as future work.

8.1 Improving the Human Interaction with Learning Techniques

Both LEARN2FIX and GRAMMAR2FIX assume that the human always provides the accurate labels of the test cases in oracle learning. However, the human could provide incorrect labels, which would reduce the quality of the automatic oracles. In future work, we will analyse the impact of mislabelling on oracle learning and automated program repair (APR). Also, we will explore techniques to deal with incorrectly labelled test cases by the human. We will look at this problem in two ways. Firstly, it is possible to minimize human errors in labelling by by distributing test cases to more than one person. The concepts of *pair programming* [200] and the work of Fabrizio et al. [61] could be useful for this task. Secondly, poorly trained automatic oracles due to mislabelled test cases can be rectified. As our automatic oracles are machine learning models, the works of Krishnan et al. [201] and Li et al. [168] could be applied to this task.

Our oracle learning techniques send artificially generated test cases for human labelling. In the experiments, we assumed that the human can comprehend all these test cases.

However, this assumption might not be true in some scenarios. In future work, we will examine the readability of the test cases presented to the human in our learning techniques. Also, we will explore how to enhance the readability of those, i.e., presenting test cases to the human in a more comprehensible manner. It will help to reduce the human effort for labelling a test case. Related to string inputs, the work of Afshan et al. [58], which is based on *natural language models*, will be helpful for this task. To extend this for other structured inputs, the ontology based method proposed by Bozkurt et al. [60] could be useful. The test data reusing method proposed by McMinn et al. [59] could be used to generate more human-readable numeric inputs.

In GRAMMAR2FIX (Chapter 5), there is no maximum limit to the human queries. Also, we observed a higher number of human queries in some subjects (Section 5.6.3). This problem can be easily resolved by distributing the labelling queries among multiple users. In future work, we will explore techniques to systematically perform this task, in which the findings of Pastore et al. [61] could be useful.

8.2 Working with Multiple Semantic Bugs

We tested our learning techniques with programs exhibiting a single semantic bug. If our learning techniques were applied to a program with multiple semantic bugs, an automatic oracle would be generated based on the collective effect of all bugs. However, it is important for developers to separately analyse the impact of each bug in a program containing multiple bugs.

Our oracle learning techniques can be improved to generate separate automatic test oracles for each bug in a program having multiple semantic bugs. We will explore how *bug isolation* approaches (e.g. Works of Zeng et al. [202], Jeremias et al. [203]) can be used for this task. Also, we will propose some adjustments to the test generation methods used in our oracle learning techniques to work with multiple semantic bugs.

When there are multiple semantic bugs in a program, the developer can separately analyse each of those if there is a separate automatic oracle for each bug. Such a setup could be useful to debug and repair multiple semantic bugs efficiently.

8.3 Using Automatic Test Oracles in Automated Program Repair

In Chapter 6, we converted the test suites generated by LEARN2FIX and GRAMMAR2FIX in oracle learning to repair test suites to be used in APR. However, we did not explore how the automatic test oracles can be applied to APR. We will explore this fact in future work.

Firstly, we will explore how to use the automatic test oracles given by our learning techniques as correctness criteria in APR. Currently, test-driven APR techniques (e.g. GenProg [102], Angelix [107]) use a repair test suite as the correctness criteria. The works such as Smith et al. [112] and Qi et al. [105] highlight the adverse impacts of this method on patch quality. We have already addressed the repair overfitting issue by using the test suites generated in our learning techniques (Chapter 6). Nonetheless, we will explore how our automatic test oracles can be used as better correctness criteria to guide APR to produce high quality patches.

Secondly, we will explore how our automatic test oracles can be applied to improve *constraint-based* program repair [9]. An automatic test oracle trained by our learning techniques describes the condition under which a semantic bug is exposed (i.e. failure condition). Therefore, a failure condition could be useful in *constraint-based* program repair to synthesise the repair constraint [9] to repair a semantic bug. We will explore this fact in future work. As our automatic oracles are machine learning models, such as binary classifiers and regular grammar, some conversions will be required to use those with APR.

8.4 Developing Specification Mining Techniques for Semantic Bugs

Specification Mining [2], i.e., inferring a formal model of behaviour from a set of observations, is useful for developers to learn about a system. Our oracle learning techniques could be applied to mine specifications for semantic bugs in a semi-automated manner. If there is a specification for a semantic bug, it is useful for developers in many ways.

An automatic test oracle trained by our learning techniques describes the condition under which a semantic bug is exposed. Therefore, it could be used as a specification of the bug. The original form of our automatic test oracles (i.e., machine learning models) might be less human-readable for developers. Therefore, we will explore how to convert our learned automatic test oracles to a human-readable form to produce bug specifications. This idea can be combined with the proposal of Section 8.2. The overall proposal will facilitate human-in-the-loop bug specification mining.

8.5 Improving Automated Debugging Techniques

As described in Section 2.5.1, *automated debugging* [74] is the other area that benefits from our study. Similar to automated program repair, automated debugging techniques such as spectrum-based fault localization (SBFL) [8, 80] and dynamic program slicing [76] could be improved with our oracle learning technique.

Both SBFL and dynamic program slicing are test-driven. The test inputs generated in our oracle learning techniques could be used as inputs to these techniques, similar to Chapter 6. Unlike in APR, it is not necessary to obtain the expected output of each test case. As our oracle learning techniques have a higher probability of generating failing tests, the accuracy of both of these debugging techniques could be significantly improved. Related to semantic bugs, this proposed method will facilitate human-in-the-loop debugging. In future work, we will experimentally analyse this concept.

8.6 Improving Greybox Fuzzing Techniques

Greybox fuzzing is an effective technique to uncover the vulnerabilities of a software system [30]. The works such as *directed gray-box fuzzing* [204] focus on guiding greybox fuzzing towards a set of target program locations. Similar to Chapter 6, we will explore methods to combine our oracle learning techniques to steer greybox fuzzers towards the faulty program behaviours of a semantic bug. The existing fuzzing greybox fuzzing mainly focuses on the bugs leading to program crashes. By combining our techniques, greybox fuzzing could be able to apply for semantic bugs. This will facilitate *human-guided greybox fuzzing* for semantic bugs.

Appendix A

Classification results of the SVM, Decision Tree and Gaussian Naive Bayes Algorithms

Table A.1: Classification Results of Support Vector Machines (SVM)

Bug	**Test suite**	**Accuracy (%)**	**Recall Failing (%)**
steve error	russtest	91.30	0
	gregtests	87.88	0
	stevetestcases	100	N/A
	stevetestcases2	100	N/A
	pjtests	97.51	0
	claytests	95	0
	dudleytests2	100	N/A
	jptests	100	N/A
	moloch tests	96.88	0
ac error	russtest	91.30	0
	gregtests	100	N/A
	stevetestcases	100	N/A
	stevetestcases2	100	N/A

Continued on next page

Table A.1 – continued from previous page

Bug	Test suite	Accuracy (%)	Recall Failing (%)
	pjtests	95.85	1.5
	claytests	95	0
	dudleytests2	No failing test cases	
	jptests	95.24	0
	moloch tests	100	N/A
bc error	russtest	91.30	0
	gregtests	100	N/A
	stevetestcases	100	N/A
	stevetestcases2	100	N/A
	pjtests	95.85	2
	claytests	95	0
	dudleytests2	No failing test cases	
	jptests	95.24	0
	moloch tests	100	N/A
ab error	russtest	91.30	0
	gregtests	84.85	0
	stevetestcases	100	N/A
	stevetestcases2	100	N/A
	pjtests	95.81	2
	claytests	95	0
	dudleytests2	90	0
	jptests	95.48	0
	moloch tests	100	N/A

Table A.2: Classification Results of Decision Trees

Bug	Test suite	Accuracy (%)	Recall Failing (%)
	russtest	83.70	50
	gregtests	85.76	25

Continued on next page

steve error

Table A.2 – continued from previous page

Bug	**Test suite**	**Accuracy (%)**	**Recall Failing (%)**
	stevetestcases	79.58	N/A
	stevetestcases2	84.17	N/A
	pjtests	96.12	40.83
	claytests	88.75	7.5
	dudleytests2	91.0	N/A
	jptests	82.62	N/A
	moloch tests	88.75	20
ac error	russtest	83.26	20
	gregtests	85.63	N/A
	stevetestcases	73.33	N/A
	stevetestcases2	72.92	N/A
	pjtests	94.90	43
	claytests	83	0
	dudleytests2	No failing test cases	
	jptests	80.24	30
	moloch tests	90.47	N/A
bc error	russtest	81.52	35
	gregtests	87.03	N/A
	stevetestcases	76.25	N/A
	stevetestcases2	76.25	N/A
	pjtests	94.63	48.5
	claytests	81.38	0
	dudleytests2	No failing test cases	
	jptests	81.90	10
	moloch tests	88.44	N/A
ab error	russtest	80.23	27.5
	gregtests	82.88	28
	stevetestcases	79.17	N/A
	stevetestcases2	85.42	N/A

Continued on next page

Table A.2 – continued from previous page

Bug	**Test suite**	**Accuracy (%)**	**Recall Failing (%)**
	pjtests	94.65	44
	claytests	83.38	0
	dudleytests2	71.5	35.0
	jptests	76.19	5
	moloch tests	89.06	N/A

Table A.3: Classification Results of Gaussian Naive Bayes

Bug	**Test suite**	**Accuracy (%)**	**Recall Failing (%)**
steve error	russtest	61.30	0
	gregtests	49.24	33.75
	stevetestcases	100.0	N/A
	stevetestcases2	92.50	N/A
	pjtests	95.31	36.67
	claytests	38.75	37.5
	dudleytests2	100.0	N/A
	jptests	22.14	N/A
	moloch tests	80.47	0
ac error	russtest	71.09	5
	gregtests	29.06	N/A
	stevetestcases	100	N/A
	stevetestcases2	64.17	N/A
	pjtests	94.79	91
	claytests	38.25	37.5
	dudleytests2	No failing test cases	
	jptests	40	70
	moloch tests	58.44	N/A
bc error	russtest	66.09	2.5
	gregtests	28.13	N/A

Continued on next page

Table A.3 – continued from previous page

Bug	**Test suite**	**Accuracy (%)**	**Recall Failing (%)**
	stevetestcases	100	N/A
	stevetestcases2	25.83	N/A
	pjtests	94.44	93
	claytests	44.38	32.5
	dudleytests2	No failing test cases	
	jptests	35.71	75
	moloch tests	59.38	N/A
ab error	russtest	68.04	0
	gregtests	56.82	29
	stevetestcases	100	N/A
	stevetestcases2	94.17	N/A
	pjtests	94.77	98
	claytests	44.63	32.5
	dudleytests2	90.0	0
	jptests	43.57	30
	moloch tests	56.88	N/A

Appendix B

Benchmarks

Benchmark	**Language(s)**	**Types of inputs**	**Number of programs**	**Number of defects**
Triangle Classification	C++	Numeric	22	18
Codeflaws	C	Numeric, string, C structure	7436	3902
IntroClass	C	Numeric, string	259	1143
QuixBugs	Python, Java	Numeric, string object	80	80

Table B.1: Benchmarks used in the experiments

Bibliography

[1] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33. ACM, 2006.

[2] 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, 2014.

[3] Ronald Rivest. Rfc1321: The md5 message-digest algorithm, 1992.

[4] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, 19(6):1665–1705, 2014.

[5] Zhiyuan Wan, David Lo, Xin Xia, and Liang Cai. Bug characteristics in blockchain systems: a large-scale empirical study. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 413–424. IEEE, 2017.

[6] A. Vahabzadeh, A. M. Fard, and A. Mesbah. An empirical study of bugs in test code. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 101–110, Sep. 2015. doi: 10.1109/ICSM.2015.7332456.

[7] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 45(1):34–67, 2017.

[8] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. Spectrum-based multiple fault localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99. IEEE, 2009.

[9] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019.

[10] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pages 309–318, 2012.

[11] Stacy J. Prowell and Jesse H. Poore. Foundations of sequence-based software specification. *IEEE transactions on Software Engineering*, 29(5):417–429, 2003.

[12] Tsong Yueh Chen, F-C Kuo, TH Tse, and Zhi Quan Zhou. Metamorphic testing and beyond. In *Eleventh Annual International Workshop on Software Technology and Engineering Practice*, pages 94–100. IEEE, 2003.

[13] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.

[14] Amanpreet Singh, Narina Thakur, and Aakanksha Sharma. A review of supervised machine learning algorithms. In *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 1310–1315. Ieee, 2016.

[15] Hu Jin, Yi Wang, Nian-Wei Chen, Zhi-Jian Gou, and Shuo Wang. Artificial neural network for automatic test oracles generation. In *2008 International Conference on Computer Science and Software Engineering*, volume 2, pages 727–730. IEEE, 2008.

[16] Meenakshi Vanmali, Mark Last, and Abraham Kandel. Using a neural network in the software testing process. *International Journal of Intelligent Systems*, 17(1): 45–62, 2002.

[17] Rushi Longadge and Snehalata Dongre. Class imbalance problem in data mining review. *arXiv preprint arXiv:1305.1707*, 2013.

[18] Tse-Hsun Chen, Meiyappan Nagappan, Emad Shihab, and Ahmed E Hassan. An empirical study of dormant bugs. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 82–91, 2014.

[19] Chris Dannen. *Introducing Ethereum and solidity*, volume 1. Springer, 2017.

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

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

[22] Lionel C Briand. Novel applications of machine learning in software testing. In *2008 The Eighth International Conference on Quality Software*, pages 3–10. IEEE, 2008.

[23] Mauro Pezze and Cheng Zhang. Automated test oracles: A survey. In *Advances in computers*, volume 95, pages 1–48. Elsevier, 2014.

[24] Paulo Augusto Nardi and Eduardo F Damasceno. A survey on test oracles. pages 50–59, 2015.

[25] Rafael AP Oliveira, Upulee Kanewala, and Paulo A Nardi. Automated test oracles: State of the art, taxonomies, and trends. In *Advances in computers*, volume 95, pages 113–199. Elsevier, 2014.

[26] Debra J Richardson, Stephanie Leif Aha, and T Owen O'malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th international conference on Software engineering*, pages 105–118, 1992.

[27] Atif M Memon, Martha E Pollack, and Mary Lou Soffa. Automated test oracles for guis. *ACM SIGSOFT Software Engineering Notes*, 25(6):30–39, 2000.

[28] Mark V Lawson. *Finite automata*. CRC Press, 2003.

[29] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. A systematic review of fuzzing techniques. *Computers & Security*, 75:118–137, 2018.

[30] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, Sep. 2018. ISSN 1558-1721. doi: 10.1109/TR.2018.2834476.

[31] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International (UK) Ltd., GBR, 1992. ISBN 0139785299.

[32] Howard Haughton and Kevin Lano. *Specification in B: An introduction using the B toolkit*. World Scientific, 1996.

[33] Fabrice Bouquet, Christophe Grandpierre, Bruno Legeard, Fabien Peureux, Nicolas Vacelet, and Mark Utting. A subset of precise uml for model-based testing. In *Proceedings of the 3rd international workshop on Advances in model-based testing*, pages 95–104, 2007.

[34] CB Jones. Systematic software development using vdm prentice hall. *Englewood Cliffs, NJ*, 1986.

[35] Bertrand Meyer. *Object-oriented software construction*, volume 2. Prentice hall Englewood Cliffs, 1997.

[36] Lilian Burdy, Yoonsik Cheon, David R Cok, Michael D Ernst, Joseph R Kiniry, Gary T Leavens, K Rustan M Leino, and Erik Poll. An overview of jml tools and applications. *International journal on software tools for technology transfer*, 7(3): 212–232, 2005.

[37] Laura K Dillon and Qing Yu. Oracles for checking temporal properties of concurrent systems. *ACM SIGSOFT Software Engineering Notes*, 19(5):140–153, 1994.

[38] Laura K Dillon and YS Ramakrishna. Generating oracles from your favorite temporal logic specifications. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 106–117, 1996.

[39] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. In *Software Engineering—ESEC/FSE'99*, pages 146–162. Springer, 1999.

[40] Roong-Ko Doong and Phyllis G Frankl. The astoot approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 3(2):101–130, 1994.

[41] Bo Yu, Liang Kong, Yufeng Zhang, and Hong Zhu. Testing java components based on algebraic specifications. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 190–199. IEEE, 2008.

[42] Pascale Le Gall and Agnes Arnould. Formal specifications and test: Correctness and oracle. In *Recent Trends in Data Type Specification*, pages 342–358. Springer, 1995.

[43] Anthony JH Simons. Jwalk: a tool for lazy, systematic testing of java classes by design introspection and user interaction. *Automated Software Engineering*, 14(4): 369–418, 2007.

[44] Algirdas Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on software engineering*, (12):1491–1501, 1985.

[45] Raúl H Rosero, Omar S Gómez, and Glen Rodríguez. 15 years of software regression testing techniques—a survey. *International Journal of Software Engineering and Knowledge Engineering*, 26(05):675–689, 2016.

[46] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability*, 22(2):67–120, 2012.

[47] Tao Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *European Conference on Object-Oriented Programming*, pages 380–403. Springer, 2006.

[48] Tao Xie and David Notkin. Checking inside the black box: Regression testing by comparing value spectra. *IEEE Transactions on software Engineering*, 31(10): 869–883, 2005.

[49] Glenn Ammons, Rastislav Bodik, and James R Larus. Mining specifications. *ACM Sigplan Notices*, 37(1):4–16, 2002.

[50] Neil Walkinshaw and Kirill Bogdanov. Inferring finite-state models with temporal constraints. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 248–257. IEEE, 2008.

[51] Marijn JH Heule and Sicco Verwer. Software model synthesis using satisfiability solvers. *Empirical Software Engineering*, 18(4):825–856, 2013.

[52] Rolf Schwitter. English as a formal specification language. In *Proceedings. 13th International Workshop on Database and Expert Systems Applications*, pages 228–232. IEEE, 2002.

[53] Mark Harman, Sung Gon Kim, Kiran Lakhotia, Phil McMinn, and Shin Yoo. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 182–191. IEEE, 2010.

[54] Javier Ferrer, Francisco Chicano, and Enrique Alba. Evolutionary algorithms for the multi-objective test data generation problem. *Software: Practice and Experience*, 42(11):1331–1362, 2012.

[55] Ramsay Taylor, Mathew Hall, Kirill Bogdanov, and John Derrick. Using behaviour inference to optimise regression test sets. In *IFIP International Conference on Testing Software and Systems*, pages 184–199. Springer, 2012.

[56] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. Efficient unit test case minimization. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 417–420, 2007.

[57] Alex Groce, Mohammed Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. Cause reduction for quick testing. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 243–252. IEEE, 2014.

[58] Sheeva Afshan, Phil McMinn, and Mark Stevenson. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 352–361. IEEE, 2013.

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

[60] Mustafa Bozkurt and Mark Harman. Automatically generating realistic test input from web services. In *Proceedings of 2011 IEEE 6th International Symposium on Service Oriented System (SOSE)*, pages 13–24. IEEE, 2011.

[61] 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. IEEE, 2013.

[62] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[63] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on software engineering*, (3):215–222, 1976.

[64] 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.

[65] Lionel C Briand, Yvan Labiche, and Zaheer Bawar. Using machine learning to refine black-box test specifications and test suites. In *2008 The Eighth International Conference on Quality Software*, pages 135–144. IEEE, 2008.

[66] Lionel C Briand, Yvan Labiche, and Xuetao Liu. Using machine learning to support debugging with tarantula. In *The 18th IEEE International Symposium on Software Reliability (ISSRE'07)*, pages 137–146. IEEE, 2007.

[67] Gary J Saavedra, Kathryn N Rodhouse, Daniel M Dunlavy, and Philip W Kegelmeyer. A review of machine learning applications in fuzzing. *arXiv preprint arXiv:1906.11133*, 2019.

[68] S Delphine Immaculate, M Farida Begam, and M Floramary. Software bug prediction using supervised machine learning algorithms. In *2019 International conference on data science and communication (IconDSC)*, pages 1–7. IEEE, 2019.

[69] Kambiz Frounchi, Lionel C Briand, Leo Grady, Yvan Labiche, and Rajesh Subramanyan. Automating image segmentation verification and validation by learning test oracles. *Information and Software Technology*, 53(12):1337–1348, 2011.

[70] Seyed Reza Shahamiri, Wan Mohd Nasir Wan Kadir, Suhaimi Ibrahim, and Siti Zaiton Mohd Hashim. An automated framework for software test oracle. *Information and Software Technology*, 53(7):774–788, 2011.

[71] Seyed Reza Shahamiri, Wan M. Wan-Kadir, Suhaimi Ibrahim, and Siti Zaiton Hashim. Artificial neural networks as multi-networks automated test oracle. *Automated Software Engg.*, 19(3):303–334, September 2012. ISSN 0928-8910. doi: 10.1007/s10515-011-0094-z. URL https://doi.org/10.1007/s10515-011-0094-z.

[72] 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*, pages 142–151. ACM, 2018.

[73] Wujie Zheng, Hao Ma, Michael R Lyu, Tao Xie, and Irwin King. Mining test oracles of web search engines. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 408–411. IEEE, 2011.

[74] Debolina Ghosh and Jagannath Singh. A systematic review on program debugging techniques. *Smart Computing Paradigms: New Progresses and Challenges*, pages 193–199, 2020.

[75] Mohammad Amin Alipour. Automated fault localization techniques: a survey. *Oregon State University*, 54(3), 2012.

[76] A. De Lucia. Program slicing: methods and applications. In *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149, 2001. doi: 10.1109/SCAM.2001.972675.

[77] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002. doi: 10.1109/32.988498.

[78] Cristian Zamfir and George Candea. Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th European conference on Computer systems*, pages 321–334, 2010.

[79] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8): 707–740, 2016.

[80] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[81] James A Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282, 2005.

[82] Manos Renieres and Steven P Reiss. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pages 30–39. IEEE, 2003.

[83] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on software engineering*, 32(10):831–848, 2006.

[84] Brock Pytlik, Manos Renieris, Shriram Krishnamurthi, and Steven P Reiss. Automated fault localization using potential invariants. *arXiv preprint cs/0310040*, 2003.

[85] Chao Liu, Xifeng Yan, Hwanjo Yu, Jiawei Han, and Philip S Yu. Mining behavior graphs for "backtrace" of noncrashing bugs. In *Proceedings of the 2005 SIAM international conference on data mining*, pages 286–297. SIAM, 2005.

[86] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for java. In *European conference on object-oriented programming*, pages 528–550. Springer, 2005.

[87] Mark Harman and Robert Hierons. An overview of program slicing. *software focus*, 2(3):85–92, 2001.

[88] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[89] Sebastian Danicic, Mark Harman, and Yoga Sivagurunathan. A parallel algorithm for static program slicing. *Information Processing Letters*, 56(6):307–313, 1995.

[90] Bogdan Korel and Janusz Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187–195, 1990.

[91] Hiralal Agrawal and Joseph R Horgan. Dynamic program slicing. *ACM SIGPlan Notices*, 25(6):246–256, 1990.

[92] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11-12):595–607, 1998.

[93] Guda A Venkatesh. The semantic approach to program slicing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 107–119, 1991.

[94] Robert J Hall. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Automated Software Engineering*, 2(1):33–53, 1995.

[95] Cyrille Artho. Iterative delta debugging. *International Journal on Software Tools for Technology Transfer*, 13(3):223–246, 2011.

[96] Ghassan Misherghi and Zhendong Su. Hdd: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*, pages 142–151, 2006.

[97] Martin Monperrus. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)*, 51(1):1–24, 2018.

[98] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 389–400, 2011.

[99] Guoliang Jin, Wei Zhang, and Dongdong Deng. Automated {Concurrency-Bug} fixing. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 221–236, 2012.

[100] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Directed test generation for effective fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 49–60, 2010.

[101] Jifeng Xuan and Martin Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 52–63, 2014.

[102] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011.

[103] Westley Weimer, Zachary P Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 356–366. IEEE, 2013.

[104] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265, 2014.

[105] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36, 2015.

[106] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: Looking for simple program repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 448–458. IEEE, 2015.

[107] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 691–701, May 2016. doi: 10.1145/2884781.2884807.

[108] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781, May 2013. doi: 10.1109/ICSE.2013.6606623.

[109] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–312, 2016.

[110] Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 727–739, 2017.

[111] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[112] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 532–543, 2015.

[113] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 831–841, 2017.

[114] Moumita Asad, Kishan Kumar Ganguly, and Kazi Sakib. Impact analysis of syntactic and semantic similarities on patch prioritization in automated program repair. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 328–332. IEEE, 2019.

[115] Ali Ghanbari. Objsim: Lightweight automatic patch prioritization via object similarity. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 541–544, 2020.

[116] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th international conference on software engineering*, pages 1–11, 2018.

[117] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. ifixr: Bug report driven program repair. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 314–325, 2019.

[118] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pages 298–309, 2018.

[119] Sungmin Kang and Shin Yoo. Language models can prioritize patches for practical program patching. In *Proceedings of the Third International Workshop on Automated Program Repair*, pages 8–15, 2022.

[120] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016.

[121] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system. *Empirical Software Engineering*, 24:33–67, 2019.

[122] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th international conference on software engineering*, pages 789–799, 2018.

[123] Burr Settles. Active learning literature survey. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2009.

[124] Alexander Clark and Shalom Lappin. Unsupervised learning and grammar induction. *The Handbook of Computational Linguistics and Natural Language Processing*, 57, 2010.

[125] Pratap Chandra Sen, Mahimarnab Hajra, and Mitadru Ghosh. Supervised classification algorithms in machine learning: A survey and review. In *Emerging technology in modelling and graphics*, pages 99–111. Springer, 2020.

[126] Abraham J Wyner, Matthew Olson, Justin Bleich, and David Mease. Explaining the success of adaboost and random forests as interpolating classifiers. *The Journal of Machine Learning Research*, 18(1):1558–1590, 2017.

[127] Vojislav Kecman. Support vector machines–an introduction. In *Support vector machines: theory and applications*, pages 1–47. Springer, 2005.

[128] L. Rokach and O. Maimon. Top-down induction of decision trees classifiers - a survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 35(4):476–487, 2005. doi: 10.1109/TSMCC.2004.843247.

[129] Samuel Kolb, Stefano Teso, Andrea Passerini, and Luc De Raedt. Learning smt (lra) constraints using smt solvers. In *IJCAI*, pages 2333–2340, 2018.

[130] Clark Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of satisfiability*, pages 1267–1329. IOS Press, 2021.

[131] Jair Cervantes, Farid Garcia-Lamont, Lisbeth Rodríguez-Mazahua, and Asdrubal Lopez. A comprehensive survey on support vector machine classification: Applications, challenges and trends. *Neurocomputing*, 408:189–215, 2020. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2019.10.118. URL https://www.sciencedirect.com/science/article/pii/S0925231220307153.

[132] Oludare Isaac Abiodun, Aman Jantan, Abiodun Esther Omolara, Kemi Victoria Dada, Nachaat AbdElatif Mohamed, and Humaira Arshad. State-of-the-art in artificial neural network applications: A survey. *Heliyon*, 4(11):e00938, 2018.

[133] Thomas G Dietterich. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*, pages 1–15. Springer, 2000.

[134] Cha Zhang and Yunqian Ma. *Ensemble machine learning: methods and applications*. Springer, 2012.

[135] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[136] Xuchun Li, Lei Wang, and Eric Sung. Adaboost with svm-based component classifiers. *Engineering Applications of Artificial Intelligence*, 21(5):785–795, 2008.

[137] Peter Linz. *An Introduction to Formal Language and Automata*. Jones and Bartlett Publishers, Inc., USA, 2006. ISBN 0763737984.

[138] Matej Črepinšek, Marjan Mernik, Barrett R Bryant, Faizan Javed, and Alan Sprague. Inferring context-free grammars for domain-specific languages. *Electronic notes in theoretical computer science*, 141(4):99–116, 2005.

[139] Baskaran Sankaran. A survey of unsupervised grammar induction. *Manuscript, Simon Fraser University*, 47, 2010.

[140] Colin De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010. doi: https://doi.org/10.1017/CBO9781139194655.

[141] E Mark Gold. Complexity of automaton identification from given data. *Information and control*, 37(3):302–320, 1978. doi: https://doi.org/10.1016/S0019-9958(78)90562-4.

[142] Haili Luo. The research of applying regular grammar to making model for lexical analyzer. In *2013 6th International Conference on Information Management, Innovation Management and Industrial Engineering*, volume 2, pages 90–92, 2013. doi: 10.1109/ICIII.2013.6703245.

[143] Katsuhiko Nakamura and Takashi Ishiwata. Synthesizing context free grammars from sample strings based on inductive cyk algorithm. In *International Colloquium on Grammatical Inference*, pages 186–195. Springer, 2000. doi: https://doi.org/10.1007/978-3-540-45257-7_15.

[144] Keita Imada and Katsuhiko Nakamura. Learning context free grammars by using sat solvers. In *2009 International Conference on Machine Learning and Applications*, pages 267–272. IEEE, 2009.

[145] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. Generating context-free grammars using classical planning. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI-17); 2017 Aug 19-25; Melbourne, Australia. Melbourne: IJCAI; 2017. p. 4391-7.* IJCAI, 2017.

[146] Siegfried Nijssen and Luc De Raedt. Grammar mining. In *Proceedings of the 2009 SIAM International Conference on Data Mining*, pages 1026–1037. SIAM, 2009.

[147] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[148] Yoon Kim, Chris Dyer, and Alexander M Rush. Compound probabilistic context-free grammars for grammar induction. *arXiv preprint arXiv:1906.10225*, 2019.

[149] Kewei Tu and Vasant Honavar. Unsupervised learning of probabilistic context-free grammar using iterative biclustering. In *International Colloquium on Grammatical Inference*, pages 224–237. Springer, 2008.

[150] Robert Giegerich. Introduction to stochastic context free grammars. *RNA Sequence, Structure, and Function: Computational and Bioinformatic Methods*, pages 85–106, 2014.

[151] William B Langdon and Riccardo Poli. *Foundations of genetic programming*. Springer Science & Business Media, 2013.

[152] Thomas E Kammeyer and Richard K Belew. Stochastic context-free grammar induction with a genetic algorithm using local search. In *FOGA*, pages 409–436. Citeseer, 1996.

[153] Ernesto Rodrigues and Heitor Silvério Lopes. Genetic programming with incremental learning for grammatical inference. In *2006 Sixth International Conference on Hybrid Intelligent Systems (HIS'06)*, pages 47–47. IEEE, 2006.

[154] Yuan Li and Jim X Chen. A nondeterministic approach to infer context free grammar from sequence. In *2014 11th International Computer Conference on Wavelet Actiev Media Technology and Information Processing (ICCWAMTIP)*, pages 1–9. IEEE, 2014.

[155] Bill Keller and Rudi Lutz. Evolutionary induction of stochastic context free grammars. *Pattern Recognition*, 38(9):1393–1406, 2005.

[156] Matej Črepinšek, Marjan Mernik, and Viljem Žumer. Extracting grammar from programs: brute force approach. *ACM Sigplan Notices*, 40(4):29–38, 2005.

[157] Olgierd Unold, Mateusz Gabor, and Wojciech Wieczorek. Unsupervised statistical learning of context-free grammar. 2020.

[158] Yifan Fu, Xingquan Zhu, and Bin Li. A survey on instance selection for active learning. *Knowledge and information systems*, 35(2):249–283, 2013.

[159] Raphael Schumann and Ines Rehbein. Active learning via membership query synthesis for semi-supervised sentence classification. In *Proceedings of the 23rd conference on computational natural language learning (CoNLL)*, pages 472–481, 2019.

[160] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987. doi: https://doi.org/10.1016/0890-5401(87)90052-6.

[161] Dana Angluin. A note on the number of queries needed to identify regular languages. *Information and control*, 51(1):76–87, 1981.

[162] Muddassar A Sindhu and Karl Meinke. Ids: An incremental learning algorithm for finite automata. *arXiv preprint arXiv:1206.2691*, 2012.

[163] Xingquan Zhu, Peng Zhang, Xiaodong Lin, and Yong Shi. Active learning from data streams. In *Proceedings of the 2007 Seventh IEEE International Conference on Data Mining*, ICDM '07, page 757–762, USA, 2007. IEEE Computer Society. ISBN 0769530184. doi: 10.1109/ICDM.2007.101. URL https://doi.org/10.1109/ICDM.2007.101.

[164] Wei Chu, Martin Zinkevich, Lihong Li, Achint Thomas, and Belle Tseng. Unbiased online active learning in data streams. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 195–203, 2011.

[165] Peng Zhang, Xingquan Zhu, Jianlong Tan, and Li Guo. Classifier and cluster ensembles for mining concept drifting data streams. In *2010 IEEE International Conference on Data Mining*, pages 1175–1180. IEEE, 2010.

[166] Alex Holub, Pietro Perona, and Michael C. Burl. Entropy-based active learning for object recognition. *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops*, pages 1–8, 2008. doi: 10.1109/CVPRW.2008.4563068.

[167] Ajay J Joshi, Fatih Porikli, and Nikolaos Papanikolopoulos. Multi-class active learning for image classification. In *2009 ieee conference on computer vision and pattern recognition*, pages 2372–2379. IEEE, 2009.

[168] Yu Li, Muxi Chen, and Qiang Xu. Hybridrepair: Towards annotation-efficient repair for deep learning models. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 227–238, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393799. doi: 10.1145/3533767.3534408. URL https://doi.org/10.1145/3533767.3534408.

[169] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[170] Robert F Woolson. Wilcoxon signed-rank test. *Wiley encyclopedia of clinical trials*, pages 1–3, 2007.

[171] Krzysztof Hryniewiecki. Basic properties of real numbers. *Formalized Mathematics*, 1(1):35–40, 1990.

[172] Russ Williams. Triangle classification problem, 2002. URL https://russcon.org/triangle_classification.html.

[173] Michal Zalewski. American fuzzy lop, 2014.

[174] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.

[175] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[176] Iqbal H Sarker. Machine learning: Algorithms, real-world applications and research directions. *SN Computer Science*, 2(3):1–21, 2021.

[177] Shin Hwei Tan, Jooyong Yi, Sergey Mechtaev, Abhik Roychoudhury, et al. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 180–182. IEEE, 2017.

[178] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015.

[179] Tae-Ki An and Moon-Hyun Kim. A new diverse adaboost classifier. In *2010 International Conference on Artificial Intelligence and Computational Intelligence*, volume 1, pages 359–363, 2010. doi: 10.1109/AICI.2010.82.

[180] Chongsheng Zhang, Changchang Liu, Xiangliang Zhang, and George Almpanidis. An up-to-date comparison of state-of-the-art classification algorithms. *Expert Systems with Applications*, 82:128–150, 2017.

[181] Anil K Jain, Jianchang Mao, and K Moidin Mohiuddin. Artificial neural networks: A tutorial. *Computer*, 29(3):31–44, 1996.

[182] Irina Rish et al. An empirical study of the naive bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46, 2001.

[183] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial machine learning at scale. *CoRR*, abs/1611.01236, 2016. URL http://arxiv.org/abs/1611.01236.

[184] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. {EcoFuzz}: Adaptive {Energy-Saving} greybox fuzzing as a variant of the adversarial {Multi-Armed} bandit. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2307–2324, 2020.

[185] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 511–522, 2013.

[186] Volodymyr Kuleshov and Doina Precup. Algorithms for multi-armed bandit problems. *arXiv preprint arXiv:1402.6028*, 2014.

[187] Luis DaCosta, Alvaro Fialho, Marc Schoenauer, and Michèle Sebag. Adaptive operator selection with dynamic multi-armed bandits. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 913–920, 2008.

[188] Joao Gama, Raquel Sebastiao, and Pedro Pereira Rodrigues. Issues in evaluation of stream learning algorithms. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 329–338, 2009.

[189] Van-Thuan Pham, Sakaar Khurana, Subhajit Roy, and Abhik Roychoudhury. Bucketing failing tests via symbolic analysis. In *International Conference on Fundamental Approaches to Software Engineering*, pages 43–59. Springer, 2017.

[190] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: A z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on*

Foundations of Software Engineering, pages 114–124, 2013. doi: 10.1145/2491411.2491456.

[191] Davide Nicolini. Zooming in and out: Studying practices by switching theoretical lenses and trailing connections. *Organization studies*, 30(12):1391–1418, 2009. doi: https://doi.org/10.1177/0170840609349875.

[192] Rahul Gopinath, Alexander Kampmann, Nikolas Havrikov, Ezekiel O Soremekun, and Andreas Zeller. Abstracting failure-inducing inputs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 237–248, 2020. doi: 10.1145/3395363.3397349.

[193] Bachir Bendrissou, Rahul Gopinath, and Andreas Zeller. "synthesizing input grammars": a replication study. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 260–268, 2022.

[194] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pages 55–56, 2017. doi: 10.1145/3135932.3135941.

[195] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru R. Căciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 47(9):1980–1997, 2021. doi: 10.1109/TSE.2019.2941681.

[196] A. Pettorossi. *Automata Theory and Formal Languages: Fundamental Notions, Theorems, and Techniques*. Undergraduate Topics in Computer Science. Springer International Publishing, 2022. ISBN 9783031119651. URL https://books.google.com.au/books?id=dT2BEAAAQBAJ.

[197] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, Sep. 2018. ISSN 1558-1721. doi: 10.1109/TR.2018.2834476.

[198] John R Koza et al. Evolution of subsumption using genetic programming. In *Proceedings of the first European conference on artificial life*, pages 110–119. MIT Press Cambridge, MA, USA, 1992.

[199] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. Semantic program repair using a reference implementation. In *Proceedings of the 40th International Conference on Software Engineering*, pages 129–139, 2018.

[200] Andrew Begel and Nachiappan Nagappan. Pair programming: what's in it for me? In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 120–128, 2008.

[201] Sanjay Krishnan, Michael J Franklin, Ken Goldberg, and Eugene Wu. Boostclean: Automated error detection and repair for machine learning. *arXiv preprint arXiv:1711.01299*, 2017.

[202] Zheng Li, Yonghao Wu, and Yong Liu. An empirical study of bug isolation on the effectiveness of multiple fault localization. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, pages 18–25. IEEE, 2019.

[203] Jeremias Röβler, Gordon Fraser, Andreas Zeller, and Alessandro Orso. Isolating failure causes through test case generation. In *Proceedings of the 2012 international symposium on software testing and analysis*, pages 309–319, 2012.

[204] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.