SCUOLA DI DOTTORATO
UNIVERSITÀ DEGLI STUDI DI MILANO-BICOCCA

Department of Informatics, Systems and Communication

PhD program in Computer Science, Cycle XXXIII

Understanding and Improving Automatic Program Repair: A Study of Code-removal Patches and a New Exception-driven Fault Localization Approach

Davide Ginelli

Registration number: 727654

Tutor: Prof. Giuseppe Vizzari

Supervisor: Prof. Leonardo Mariani

Coordinator: Prof. Leonardo Mariani

ACADEMIC YEAR 2019/2020

UNIVERSITY OF MILANO - BICOCCA

Abstract

School of Science
Department of Informatics, Systems and Communication

Doctor of Philosophy

Understanding and Improving Automatic Program Repair: A Study of Code-removal Patches and a New Exception-driven Fault Localization Approach

by Davide GINELLI

Debugging and bug fixing are extremely important activities that are regularly performed to eliminate defects from software. They are however time consuming, and thus improving their degree of automation is increasingly important in a competitive world. A possible solution is offered by Automatic Program Repair (APR) techniques, through which it is possible to automatically generate patches that can be either presented to developers as candidate patches or directly integrated into the target programs. Although many different APR techniques have been developed in the last few years, there are still open challenges related to their introduction as stable and working solutions in development pipelines. Indeed, most of the APR techniques rely on test cases to evaluate the correctness of patches, which is a weak validation method that can lead to the generation of incorrect patches. In particular, recent empirical studies show that APR techniques often result in the generation of code-removal patches, that are patches that drop functionalities to address the faults that afflict programs. Yet another aspect that strongly influences the success of APR is fault localization. Indeed, if the correct location to generate the patch is not found, it is hard or even impossible to generate a patch. Experimental evidence shows that current strategies used for the fault localization are often unable to identify the correct statements to be modified, making the generation of patches extremely hard. In this context, this Ph.D. thesis provides two key contributions: 1) an empirical study about the factors that influence the generation of code-removal patches and an analysis of the useful information that can be extracted from them; and 2) a new fault localization technique that exploits the semantic of exceptions to accurately guide the fault localization process.

Acknowledgements

I would like to thank my supervisor, Prof. Leonardo Mariani, who guided me during the Ph.D., inspiring me with his passion for research and helping me along the way.

I also want to thank the research group that I worked with, supporting me in the various stages of my Ph.D.

I would like to thank my friends, with whom I shared joys and disappointments, and who contributed to make this experience unforgettable .

Finally, a special thank you to my family, that was always with me.

Contents

List of Figures

List of Tables

Introduction

Software is everywhere and plays an important role in the daily life of every person. It is present in smartphones, TVs, cars, vending machines, and in many other systems designed to help and simplify the life of people. Nowadays, a single application is enough to carry out several different tasks, such as finding paths, turning on the heating system before going home, or buying items online.

The society is highly and dangerously dependent on software. Indeed, software may have problems related to defects not discovered before its release, or also related to human errors not necessarily made during the developing phase.

For example, in May 2015, a bug in iOS allowed iPhone users to make other iPhones rebooting and shutting down continuously by simply sending a text message containing a string of specific Arabic characters, the meaning of which was "effective. Power". The crash was caused by the way Unicode characters are decoded, which overloaded the device's memory [28].

An example of software failure caused by accidental errors is the one occurred to the Amazon S3 Service on February 28th, 2017. In that occasion, the developers that were debugging an already present issue executed a command with the intention to remove a small number of servers related to the issue, but one of the inputs to the command was wrong and a larger number of servers was removed, causing malfunction to the applications based on the Amazon cloud infrastructure [4].

Thus, the activities of testing and debugging are extremely important to try to minimize the presence of defects and to avoid their reappearance. On the other hand, these types of activities are time consuming, and this is a critical aspect in an increasingly competitive world, where it is important to always have a stable version of the software, in order to avoid the users dissatisfaction, and to have more time to develop new features and release them in short development cycles.

Nowadays, to be able to address these types of requirements, the automation of the different phases involved in the traditional software development life cycles has a key role. In this regard, development practices like Continuous Integration (CI) and Continuous Delivery (CD) can be used to build and run test cases to find and address bugs in a faster way than the past, and to deploy and release software in production safely and quickly with minimal human intervention.

However, also when exploiting CI and CD, developers have often to work on faults, to identify their root causes and understand how to fix them. This activity might be challenging and time consuming, because it requires the analysis and understanding of potentially complicated executions. In this regard, a possible solution is offered by Automatic Program Repair (APR) techniques, through which it is possible to automatically generate patches that can be integrated into the target programs or suggested to developers as candidate patches. Thanks to APR techniques, developers can reduce the time usually allocated to understand the problem and localize where and how to change the program to produce a patch. In this way, it is possible to speed up the repair process, alleviating the developers' effort in testing and debugging activities, and letting the developers to allocate their effort more effectively on other tasks.

Research in APR is particularly active. Although many different techniques have been developed in the last few years, there are still open challenges related to the introduction of APR techniques as stable and working solutions in development pipelines. Indeed, most of the APR techniques rely on test cases to evaluate the correctness of patches, but this is a weak validation method that can lead to the generation of test-suite-adequate patches that are incorrect. In particular, recent empirical studies show that APR techniques often result in the generation of code-removal patches [95, 69], that are patches that drop functionalities to address the faults that afflict programs. Another aspect that strongly influences the success of the generation of a patch is related to the fault localization. Indeed, if the correct location to create the patch is not found, it is hard or even impossible to generate a patch [66]. Experimental evidence shows that current strategies used for the fault localization are not able to rank the suspicious statements at top positions, making the generation of patches extremely hard.

In this context, this Ph.D. thesis provides two key contributions:

- An empirical study about the factors that influence the generation of code-removal patches and an analysis of the useful information that can be extracted from code-removal patches;
- A new fault localization technique that exploits the semantic of the exceptions to accurately guide the localization process.

This Ph.D. thesis is organized as follows: Chapter 1 introduces software testing, debugging, and automatic program repair. Chapter 2 discusses state of the art APR techniques and their limitations. Chapter 3 presents the results obtained with an empirical study on the effectiveness of code-removal patches. Chapter 4 presents a new approach to fault localization that exploits the semantics of exceptions. Finally, Chapter 5 provides final remarks, summarizing the main findings and the open challenges.

Chapter 1

Software Testing, Debugging and APR in the Development Pipeline

This chapter introduces software testing and debugging, providing definitions and describing the main approaches. It finally describes some of the challenges related to testing and debugging activities, and discusses how automatic program repair can be introduced in the software development pipeline to support developers.

1.1 Software Testing

Software testing is one of the most relevant activities concerning software development.

Definition 1.1.1 (Software testing)**.** Software testing is the process related to the execution of a program or a system in order to reveal faults, and it comprises any activity that aims to evaluate an attribute or a capability of a program to determine if the required results are satisfied [85, 36, 89].

Definition 1.1.2 (Fault or Defect)**.** A fault (or defect) is an anomaly in the software that may cause an incorrect behavior. It is also possible to use the term *bug* to indicate a fault [12].

A fault in a program is a direct consequence of an error made by a developer.

Definition 1.1.3 (Error)**.** An error is a human action that produces an incorrect result. For example, a developer might make a mistake when typing a variable name [12].

Although a program may contain a fault, it could run for a long period, without showing any incorrect behavior. Indeed, to ensure that this happens, it is necessary that three conditions occur: 1) the input of the software must exercise the faulty statement, 2) the faulty statement must produce a different result compared to the correct one, and 3) this wrong result must propagate to the output, so that it is possible to detect the wrong behavior [116].

When these tree conditions are satisfied, it means that a software failure occurred.

Definition 1.1.4 (Failure)**.** A failure is the inability of a system or component to perform its required functions within specified performance requirements [12].

Verification and Validation

Software testing embraces two different processes: 1) **verification**, and 2) **validation**. The first one is related to the evaluation of a software (or its component) to determine if the products of a certain development phase satisfy the conditions established at the beginning of that phase. This process is usually performed inspecting and reviewing the software deliverables [12]. The second one is related to the evaluation of a software (or its component) in order to establish if it satisfies the specified requirements. This process is usually performed executing test cases [12]. Thus, through the verification process, it is possible to verify if the specifications defined by the developers about how the program should be are correctly implemented, while through the validation process, it is possible to establish if developers are implementing the right program that satisfies the user's needs defined on the basis of the agreed requirements [93]. If the specifications are wrong, the verification process cannot help the developers to detect if the program does what the stakeholders really want.

The test cases used in the validation process are characterized by three aspects: 1) a **set of test inputs**, that are the data received from an external source (e.g., hardware or humans) used by the code under test, 2) the **execution conditions**, that represent the conditions under which to run the test cases (e.g., environment settings, as a specific version of a database), and 3) the **expected outputs**, that represent the results that the program under test should produce.
To determine if a program produces an incorrect behavior, it is necessary to use a test oracle.

Definition 1.1.5 (Test oracle)**.** A test oracle is a document or a piece of software with which it is possible to establish if a test case passes (correct behavior) or fails (wrong behavior) by comparing the expected output with the output produced by the program under test [12, 103].

A common way of implementing oracles is by adding assertions in the test cases.

Definition 1.1.6 (Assertion)**.** An assertion is a formal statement usually implemented as a Boolean expression that must be evaluated to true to make the test pass [103]. A test case can be characterized by one or more assertions. If it has more than one assertion, all the assertions must be evaluated to true to make the test pass.

Levels of Testing

Based on the element under test, it is possible to distinguish three main categories of tests: 1) unit tests, 2) integration tests, and 3) system tests.

Unit Test The main goal of a unit test is to ensure that each individual unit of a software is working according to its specification, where a unit is the smallest possible testable software component. It is possible to test different features of each unit, such as functions, performance requirements, states, states transitions, control structures, and data flow patterns. Usually, considering procedural languages, a unit is a function or a procedure; considering object-oriented languages, a unit can be both a method or a class [12].

```java
1  import static org.junit.jupiter.api.Assertions.assertEquals;
2  import example.util.Calculator;
3  import org.junit.jupiter.api.Test;
4  
5  public class ExampleTest {
6  
7      private final Calculator calculator = new Calculator();
8  
9      @Test
10     void addition() {
11         assertEquals(3, calculator.add(1, 2));
12     }
13 }
```

LISTING 1: Example of Unit Test.

Listing 1 shows an example of unit test. In this case, the aim of the test is to evaluate if the method `add` of the class `Calculator` behaves correctly. In particular, the assertion at line 11 verifies if the expected output (3) is equal to the one computed with the method `add` under test, that receives as input two integer values (1, and 2).

Integration Test The goal of an integration test is to evaluate the interaction of components, that are tested as a group. In general, once a unit passed unit testing, the unit is integrated with the set of the previously integrated modules, so as to evaluate the subsystem [12]. Once all the units are integrated successfully, it is possible to proceed with the system tests.

System Test The goal of a system test is to evaluate the system as a whole, considering both functional behaviors and quality requirements, such as reliability, usability, security, and performance. System testing can detect defects related to external hardware and software interfaces, like the ones that cause deadlocks or problems with an ineffective usage of the memory [12].

Test Automation

Interestingly, test execution can be automated. Through the use of automated tools for testing, it is possible to increase the productivity, to shorten the time of the different stages of development, to reduce the risks, and to improve both the software product and the process quality [12]. Once created, tests can be run over and over, every time that there is a change in the program, without manual effort and in a faster way compared to the manual tests.

For instance, JUnit, Mockito, Hamcrest, Selenium, and Appium are among the most used libraries for Java.

JUnit [113] is an open source unit testing framework, designed with the purpose to write repeatable tests.

Mockito [110] is an open source mocking framework to write JUnit test cases. A mock object is an implementation for an interface or a class, that simulates their behavior, allowing to set the output of certain methods. For example, to simulate the source of a data, through Mockito it is possible to create a mock that works as data provider, ensuring that the test conditions are always the same.

Like JUnit, Hamcrest [34] has been designed to implement unit tests. In particular, it offers the possibility to define customized assertion matchers to check certain conditions.

Selenium [102] is a suite of tools for automating web browsers. It provides extensions that allow the emulation of user interactions with browsers, and a distribution server for scaling browser allocation. It is designed to support functional tests, such as unit, integration, and usability tests.

Appium is an open source test automation framework compatible with native, hybrid and mobile web apps.

Possible Results of Test Cases

If a test case fails to produce the expected output, developers have to investigate the cause of the failure. When a test fails, it is possible to distinguish two main cases: failing test cases and crashing test cases.

Definition 1.1.7 (Failing test case)**.** A failing test case is a test case that fails due to a violated assertion. It means that the actual values generated by the program under test violate one of the assertions present in the tests. Per the terminology of JUnit, this is a test failure [56]. Thus, a failing test case reports an invalid test result.

Definition 1.1.8 (Crashing test case)**.** A crashing test case is a test case for which the program under test generates an exception during its execution. The exception can be caught or uncaught. In the former case, the exception is caught using a `try-catch` statement in the test case that, for example, uses the instruction `Assertion.fail()`[1] in the body of the `catch` to make the test fail. In the latter case, the execution terminates with an uncaught exception, per terminology of JUnit, causing a test error [56]. A frequent case is the case of the program raising a `NullPointerException`[2], which occurs when a program attempts to use an object that has not a value.

[1] `https://junit.org/junit4/javadoc/4.12/org/junit/Assert.html#fail()`

[2] `https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/NullPointerException.html`

Considering Listing 1, the test `addition` is a *failing test case* if the assertion at line 11 is not satisfied, that is the result of the instruction `calculator.add(1, 2) is not equal to the expected value (3)`. On the other hand, the test `addition` is a *crashing test case* if during the execution of the instruction `calculator.add(1, 2)` the program throws a `NullPointerException`, and thus the assertion is not evaluated.

1.2 Software Debugging

A test failure normally requires developers to investigate the test and the program to identify its causes. This activity is referred to as debugging.

Definition 1.2.1 (Software debugging)**.** Software debugging is the process related to localize the defects, repairing the code, and restarting the software [12].

There are several techniques that can be used in the debug process to help developers when there is a failing or crashing test case in a program. In particular, some of the most used are: Trace-based debugging, Spectrum-based debugging, and Delta debugging.

Trace-based Debugging Trace-based debugging is based on the concept of *breakpoint*, that is a point in the source code set by the developer in which the program stops its execution, so that it is possible to analyze its state in that moment (e.g., the value of a variable). After reaching the breakpoint, the developer can examine the state of the program executing it line by line, or passing between the invoked functions. The limitation of this approach is that if a developer chooses a line of code that is not related to the fault, she will not be able to get useful information to address the fault. In this regard, Whyline is a debugging tool proposed by Ko and Myers, that allows developers to select a question, instead of a line code, about the program output [49]. In this way, developers can focus directly on the questions related to the behavior of the program, and not on choosing which lines of code are useful to investigate.

Spectrum-based Debugging Spectrum-based Debugging, also known as Spectrum-based Fault Localization, uses the result of test cases and the information about the executed statements in order to compute the likelihood that an entity of a program (e.g., a statement) is faulty [114]. The techniques that belong to this category can follow different strategies to find the bugs. The majority of them rank the suspicious entities of a program by using specific formulas, based on spectrum information derived by the execution of test cases. The idea shared by most formulas is that program entities that are executed more often by failing test cases and less often by passing test cases have a higher suspiciousness score, and thus are more likely to be faulty [20]. Testers are supposed to follow the ranking when inspecting the program looking for the fault that originated a failure.

Delta Debugging

Delta Debugging approaches debug by simplifying the failing test case, turning it in a *minimal* test case, that is a test case that only includes the inputs that are relevant to reproduce the failure. This means that a failing test case is considered *minimal* when any code element cannot be removed without making the test unable to reveal the failure. Delta debugging is fully automated, indeed whenever a regression test fails (i.e., a test case already executed in a previous phase with success, starts to fail after a new change in the program), the algorithm can automatically determine the circumstances that induce the failure [137].

1.3 Bug Fixing and Modern Development Models

Besides the technical aspects of software testing, like techniques, methods, and tools used to evaluate the quality of software, there is the economic aspect that plays an important role. The cost of software failures is often significantly high. For instance, a report conducted by the Consortium for IT Software Quality (CISQ) indicates that the cost of poor-quality software in the US in 2018 has been approximately \$2.84 trillion, with 37.46% due to software failures, and 16.87% due to activities performed to find and fix bugs [53]. Similarly, multiple studies show that developers can spend up to 75% of their time in debugging and fixing activity [11, 108, 16].

Thus, software testing and debugging are time consuming activities that can be a problem in an increasingly competitive world, where it is important to always have a stable, working, and updated software, in order to avoid the users dissatisfaction, possible loss of money, and to have more time allocated for the development of new features and releasing them in short time periods.

Nowadays, to be able to satisfy these requirements, a key role is represented by the automation of the different phases involved in the traditional software development life cycles.

Definition 1.3.1 (Software Development Life Cycle)**.** Software Development Life Cycle (SDLC) is the process related to the build and maintenance of a software system. Usually, it comprises different phases, starting from the requirement analysis to the testing and evaluation phase [63].

The approach that best represents the idea to automatize every phase is DevOps.

1.3.1 DevOps

Definition 1.3.2 (DevOps)**.** DevOps can be defined as an approach to software development used by both developers and operations teams to build, test, deploy, and monitor programs, ensuring speed, quality, and control [39].

The idea of DevOps is to integrate the two often separated worlds of *Development* and *Operations* using automated development, deployment, and infrastructure monitoring, with the aim of delivering software faster and continuously, reducing problems related to misunderstandings between team members [26].

DevOps implementations generally rely on an integrated set of tools to remove the manual steps, and thus reducing the errors. In particular, considering testing and debugging activities, there are two main practices in the software development for which the automation plays an important role to help the developers: 1) Continuous Integration (CI), and 2) Continuous Delivery (CD).

The first one can help in finding and eliminating the bugs early in the development cycle, and the second one ensures that the software can be reliably released at any time.

Continuous Integration

In the past, developers were used to work on their local copy of the project and merge the changes only at the end. This behavior made the operation of merging difficult and time consuming due to the conflicts between the different changes applied by the developers.

To avoid this problem, modern software development can rely on CI, that is a software development practice in which the software is built and tested every time a change is applied to the application [32].

With CI, developers frequently commit the changes to a shared repository using a version control system. Then, the CI service automatically builds and runs units test cases to detect any errors.

The main goals of CI are to find and address bugs in a faster way than the past, improve the software quality and reduce the time needed for the validation and releasing of software updates [9].

Some of the most used CI services are Jenkins[3], Travis CI[4], that can be used by projects hosted at GitHub and Bitbucket, and GitLab CI[5], that can be used by projects hosted on GitLab.

Continuous Delivery

CD is a software engineering approach in which CI, automated testing, and automated deployment capabilities allow software to be deployed and released in production safely and quickly with minimal human intervention [38, 32].

With CD, every code change is built, tested, and then pushed to a non-production testing or staging environment.

The automation of tests is focused to verify how a software feature/update works considering different aspects that go beyond the unit tests, such as the usability of the user interface, the integration with other modules of the software or external services, and the reliability of the API. The final decision to deploy the artifact to

[3] `https://www.jenkins.io`
[4] `https://travis-ci.org`
[5] `https://about.gitlab.com/stages-devops-lifecycle/continuous-integration/`

a production environment is triggered by the developers [8]. If the deployment is done without the explicit intervention of a human, the expression that is used is *continuous deployment*, and not continuous delivery.

1.4 APR in the Software Development Pipeline

Debugging techniques can provide useful information about the possible locations of faults and the states of the application in relation to a failure, but still significant effort is required to developers for analyzing this information to identify the faults and understand how to fix them. Considering both the user experience and losses of money that a fault can potentially cause, it is important to be able to fix a fault in a fast way, as soon as it is detected. Moreover, it is important also to have a system that allows to detect and fix the defects without consuming too many resources of the developers.

CI and CD represent a working solution to help the developers in constantly monitor software in order to detect as soon as possible any faults, but a lot of time is still necessary by developers to identify the root cause of the fault and understand how to fix it.

A new solution is represented by Automatic Program Repair (APR) techniques, which are able to automatically suggest patches [30, 81]. The idea at the basis of these techniques is to try to automatically fix the faults in a program, reducing developers' effort. Indeed, the patch both explains the reason of the failure and provides a possible solution to the problem. Thus, even if a change proposed by an APR tool is incorrect, developers can exploit it to have more details about the fault and reduce the time needed to find the error and to fix the bug [30].

The research in this field is currently active, and many approaches and techniques have been designed and developed. Recent studies are trying to integrate the APR tools in the development pipeline, with the idea to have a bot that constantly monitors the build status in the CI system, and tries to create a patch for every build failures using different repair techniques [10, 115]. Even outside the academia, there are solutions in this direction, like Dependabot[6], that is a bot integrated in GitHub that checks for outdated or insecure dependencies in projects and it automatically creates a pull request to update every dependency that does not pass its security check. In this way, the developers can review the pull request and decide if to merge the changes proposed by the bot or not.

Despite the numerous studies in the field of APR, there are still open challenges.

Overfitting Patches A key challenge is related to the correctness of patches. Even a patch that passes all the available test cases is not always good enough to fix bugs, due for instance to weaknesses of the test suite. As a consequence, a synthesized

[6] `https://dependabot.com`

patch may introduce new problems not discovered by the test suite [95, 132], sometime even removing faults by removing the entire functionality that includes the faulty code [31]. These types of problematic patches are called *overfitting* patches, and they are discussed in detail in Section 2.2.

Fault Localization Another challenge is related to the localization of the faults to be fixed [66], since current repair techniques are good in modifying programs, but are not always effective in the localization of the faulty statements [66, 6, 91].

This Ph.D. thesis focuses on the analysis of these open challenges, providing new findings and possible solutions. In particular, Chapter 2 describes the main approaches in the field of APR, and analyzes in detail the problem of overfitting patches and the difficulties related to the fault localization. After explaining more in detail APR and these limitations, Chapter 3 investigates the effectiveness of code-removal patches, that are among the most common and problematic type of overfitting patches. The chapter shows how these patches may be generated, identifies the reason why they pass test cases, and discusses how they can be exploited by developers even when they are not correct. Chapter 4 describes the current limitations of the strategies used by APR techniques to localize faults, and proposes a new approach that exploits the semantic of the failure to guide the localization process.

Chapter 2

Automatic Program Repair

This chapter provides an overview of the main approaches in the field of Automatic Program Repair (APR), and discusses two of the main limitations of program repair techniques: the generation of overfitting patches and the challenge of fault localization.

2.1 Overview of Automatic Program Repair Techniques

APR techniques can automatically generate patches that can be integrated in the target programs or suggested to developers as candidate patches. In the following, the description of how the repair process works and the main related approaches are provided.

2.1.1 Program Repair Process

As shown in Figure 2.1, program repair can be defined as a process that receives a program with at least a failing or crashing test case in input and produces either a patched program or no patch in output. The first step of the process consists of the localization of the locations where the patch could be applied. As described in Section 2.3, there are different strategies for fault localization, and the most widely used techniques are *spectrum-based* fault localization (SBFL) solutions, which rank the suspicious statements by suspiciousness based on test coverage information.

Once the set of candidate locations is available, the repair process tentatively modifies the program according to a strategy. It can 1) generate a candidate solution (i.e., a new variant of the program), and then validate the candidate solution using the available test cases (i.e., the program is patched if it passes all the test cases), or 2) formally encode the program repair problem such that a solver can be used to obtain a patched program. This part of the process can be repeated multiple times for multiple program locations, according to the output of fault localization. This process continues until 1) a patch is found, 2) no other patches can be produced, or 3) the time allocated for the repair process expires [30].

It is possible to distinguish three repair strategies based on how the faulty program is defined and the fault is addressed: 1) *generate-and-validate*, 2) *semantics-driven*, and 3) *learning-based repair* techniques [30, 62].

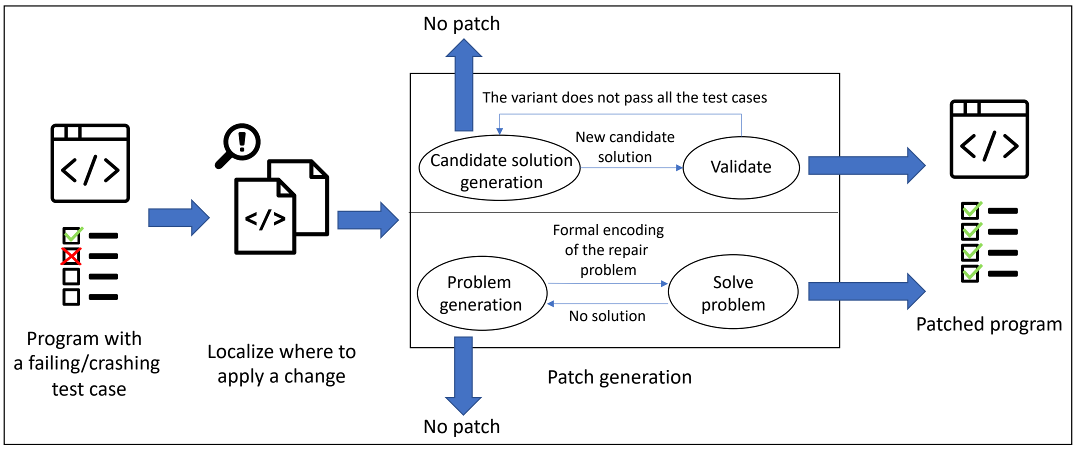

No patch
The variant does not pass all the test cases
New candidate solution
Candidate solution generation
Validate
Formal encoding of the repair problem
Problem generation
Solve problem
No solution
Program with a failing/crashing test case
Localize where to apply a change
Patch generation
No patch
Patched program

FIGURE 2.1: Description of the Program Repair Process.

2.1.2 Generate-and-validate Techniques

Generate-and-validate techniques follow an iterative process characterized by two activities: 1) *generate* activity that produces the candidate solutions, and 2) *validate* activity that verifies if the candidate solutions are correct (i.e., they pass all the test cases) or not [30].

The *generate* activity uses a set of *change operators* to modify the original buggy program, and it produces new variants of the same program, that are called *candidate solutions*. There are different types of change operators: 1) atomic change operators, that change a program in one single point, 2) predefined template operators, that change a program according to some predefined strategy, and 3) example-based template operators, that change a program following the example of what has been done in the past by the developers to fix a similar fault.

The *validate* activity uses the available test cases to establish if a candidate solution produced during the *generate* activity passes all the test cases, and thus it can be considered a possible patch or not.

The generate and validate activities can be executed according to two main strategies: 1) *search-based*, that applies the change operators randomly or following a certain heuristic search algorithm, and 2) *brute-force*, that systematically produces every possible change using the chosen set of operators on the suspicious points identified by fault localization.

Techniques that use Atomic Change Operators

Atomic change operators modify a program in one single point of its representation, that is the Abstract Syntax Tree (AST). These operators can add, delete, or modify a node in the AST, that can correspond to an entire statement or part of it, such as a variable or an operator [30].

Search-based Techniques The techniques belonging to this category use a randomized search process to potentially manage any type of fault.

One of the first tool that implements this approach is GenProg [61], that exploits the genetic programming in order to guide the repair process. This process consists in one ore more iterations in which every time an atomic operator is applied in a location based on its suspiciousness as described in Section 2.3.1. Moreover, if after an iteration no patches are found but there are at least two candidate solutions, GenProg exploits *single-point crossover*. Single-point crossover consists in randomly taking two candidate solutions (A and B) and randomly selecting a program point per solution to produce two new candidate solutions by merging the initial part of A with the final part of B, and vice versa. GenProg validates every candidate solution using the test suite and using a *fitness function*, that allows to discard or to keep a candidate solution for the next iteration based on a score computed taking into consideration the number of passed and failed test cases.

A similar technique to GenProg is Marriagent [50]. The difference is related to the way in which the crossover is applied. Indeed, Marriagent chooses the candidate solutions to be merged based on their difference. The more is the difference expressed as the changes applied to the original program, the higher is the probability to merge two candidate solutions. The idea behind this strategy is that selecting candidate solutions that are similar has more likelihood to create new solutions with similar results, that thus do not improve the quality of the previous solutions.

Another technique based on genetic programming is pyEDB [3]. Its peculiarity is related to how it represents a candidate solution. Indeed, while GenProg uses a full representation of the source code associated with a candidate solution, pyEDB represents a candidate solution as the set of changes applied to it compared to the original version of the program. This representation is more compact to the one used by GenProg and thus it is easier to handle and it requires less memory consumption.

Also MUT-APR [7] works in the same way of GenProg, but it is able to change only the operators in a program, thus it can fix only faults that are related to a wrong use of arithmetic, relational, bitwise, and shift operators.

The genetic programming is not the only solution adopted by the search-based techniques. Indeed, instead of using the evolutionary approach of genetic programming, RSRepair [94] is an example of tool that uses a random search, and prioritizes the execution of the test cases that are able to discard more candidate solutions. The idea is that these tests could have more likelihood to faster detect if a candidate solution has to be discarded or not, so as to speed up the repair process. This technique can apply only one single change to every single candidate solution. If this passes all the test cases, it is considered a possible patch, otherwise it is discarded.

An extension of RSRepair is SCRepair [40], that uses a particular metric to guide the selection strategy of a piece of code that can be used to replace another one that is supposed to be faulty. In particular, the approach searches for pieces of code that are not identical to the code that has to be replaced, but that can be integrated well with code around the location that needs the change. The idea is that similar code fragments may contain the ingredients to create the patches.

To increase the randomness of selecting a location in which to apply a patch, there are approaches like JAFF [5] that randomly selects *n* locations, and then it selects only the ones that have an higher suspicious score.

Finally, unlike the other techniques that focus only on changing the source code of the program, CASC [123] is a program repair tool that evolves not only the source code, but also the test cases. The idea is that evolving the test case allows to better find defects in the programs, and also to discard program variants that are not correct.

Brute-force Techniques The techniques belonging to this category explore the search space systematically, that is they try to create a patch changing a point in a program applying every possible type of change.

PACHIKA [18] is a program repair technique which aim is to infer the preconditions that have to be satisfied by the methods called during the execution of a program, so that the program passes all the test cases. The technique works only if the failing test case that reveals the fault violates one or more of these preconditions. The approach systematically adds or removes the method calls to influence the violated preconditions so as to find one version of the program able to pass all the test cases.

AE [119] systematically applies one single change to a single statement. Its main characteristic is the reduction of the search space by discarding the candidate solutions that are semantically equivalent to others although syntactically different. In this way, it is possible to speed up the evaluation process, since it is not necessary running again the test cases for these candidate solutions.

Kali [95] is another technique that tries to fix a program by systematically deleting functionalities. The removal of a functionality is performed by 1) changing an if condition in order to avoid the execution of a specific path of the program, 2) by removing a statement, or 3) by adding a return statement that allows to avoid the execution of the subsequent statements. This approach has been designed by the observation that the majority of the patches produced by systems like GenProg [61], RSRepair [94], and AE [119] consist in a deletion of a single functionality. The authors showed the effectiveness of this system and pointed out the attention about the fact that test suite are a too weak proxy to be used in the evaluation of patches produced by program repair systems. In this thesis, the effectiveness of code-removal patches has been in-depth investigated and the results are available in Chapter 3.

Techniques that use Template-based Change Operators

These techniques try to create a patch modifying a program in one or more locations following predefined templates based on the type of fault. For example, a template could be a set of changes that add specific code to manipulate the program conditions. Most of these techniques use the brute-force approach instead of the search-based one, because the application of templates can be very expensive in terms of time, reducing the speed of the evolution of program variants performed in the search-based approach [30].

Search-based Techniques In this area, it is possible to find techniques specialized only for specific types of faults. For example, ARC [47] is a technique designed to address concurrency faults. The approach consists in evolving the buggy program using genetic programming and a set of templates that allow to perform changes related to the concurrency, such as the addition of piece of code to synchronize unprotected shared resources.

Brute-force Techniques The repair process performed by these techniques consists in applying every possible template to every location until the patch is created or the time allocated for the repair activity ends.

AutoFix-E [118] is a program repair technique that works with programs written in Eiffel, a programming language that supports the use of contracts. This technique aims to create a patch applying a set of templates so that the program does not violate none of the contracts defined in it. These templates are 1) the addition of a new piece of code before the one related to the fault, 2) the addition of an if-statement before the faulty statement to execute a new piece of code if a contract is violated, 3) the addition of an if-statement to execute the old piece of code if the contract is not violated, and 4) the addition of an if-else statement, that allows to execute a new piece of code if the conditions of the contract are violated, and the old code vice-versa.

AutoFix-E2 [92] is an evolution of AutoFix-E that improves the way in which a template is applied exploiting not only the information extracted from the contracts, but also the one extracted from the conditions evaluated during the execution of the test cases.

Another example of these type of techniques is SPR [70], that exploits a set of parameterized templates with the aim of synthesizing a condition that allows to assign the values to the parameters of the applied template making the program passing all the test cases.

Techniques that use Example-based Template Operators

The techniques that belong to this category modify a program using a set of templates extracted from patches produced in the past. The extraction of the templates can be done manually or automatically using for example mining techniques [30].

Search-based Techniques In this area, the templates are automatically recombined by search-based algorithms with the aim to maximize their application in the repair process. The idea is that some faults might require the application of different templates to create the patch [30].

History-driven repair [59] uses the evolutionary approach of GenProg, without using the concept of crossover, and it extracts patch patterns considering the patches developed in the past for different projects to guide the repair process.

PAR [48] exploits templates defined after performing a manual analysis of more than 60,000 real-word patches. These templates are encoded as a sequence of rewriting rules applied to the Abstract Syntax Tree of a program.

A similar approach is implemented in Relifix [111], that is a technique designed to address regression problems. Also in this case, to create a patch, it uses a set of patterns extracted by the manual analysis of 73 real regression faults, such as replacing a statement with the previous version.

Brute-force Techniques The techniques belonging to this area can automatically extract templates every time from a different set of samples [30]. They can address general types of faults or specific ones, such as buffer overflow problems.

An example of generic technique is R2Fix [64], that exploits the bug reports filed by the users and a set of predefined templates associated with the patches for those bug reports. Using the machine learning, the technique identifies the bug reports that are similar to the bug report of the fault that has to be fixed, and then it systematically applies the templates associated with the identified bug reports to the program, contextualizing them to the code of the new fault.

On the other side, CodePhage [104] is an example of technique specifically designed to address buffer overflow problems. This approach uses a set of *donor* programs, that are programs that implement the same functionality of the faulty one, to determine the conditions that should be used to fix the buffer overflow problem in the program under repair. The idea is that at least one of the donor programs might have the check that is missing in the faulty program, and the addition of this check would repair the program.

2.1.3 Semantics-driven Techniques

Semantics-driven techniques encode the program repair problem formally either explicitly using for example a formula which solutions correspond to the possible patches of the program under repair or implicitly as an analytical procedure which outcome is a patch [30].

SemFix [86] is a program repair technique which aim is to synthesize a patch through the change of a branch predicate or through the change of the right part of an assignment. The tool implementing this technique replaces the expression associated with the branch predicate or with the right part of the assignment with a symbolic expression, and after that it runs the program using the test cases in order to get a set of constraints for the symbolic expression that makes the program pass all the test cases. During the execution, the program is run concretely until reaching the symbolic expression, and then it is executed symbolically. Using the constraints and a set of components (e.g., arithmetic operators or logical operators), SemFix tries to synthesize the patch. This technique can create patches only for faults that do not require changes in multiple points.

DirectFix [80] translates the faulty program in a *trace formula* f to encode its behavior, and it translates the failed test cases into a set of *oracle constraints* O. To synthesize the patch, the idea behind DirectFix is to modify the formula f so that putting it in conjunction with O, it is satisfiable. The repair process consists in reducing the logic problem into an instance of the partial MaxSAT problem, and in using a partial MaxSMT solver to generate a new formula f' that is able to satisfy the condition $f' \wedge O$. The patch is possible only if a solution to this problem is found.

Angelix [79] exploits the concepts of *angelic path* and *angelic forest* to guide the repair process. The first one is used to encode part of the repair problem as a set of triples, where each of them is characterized by: 1) the suspicious expression, 2) the value that the expression should return to pass the test cases (it is called *angelic value*, and 3) the set of variables that are in the scope of the suspicious expression (this set is called *angelic state*). The concept of angelic forest allows to encode the repair problem as a set of angelic paths and is passed to a patch synthesis engine to produce a patch that can consist of multiple lines.

SearchRepair [46] uses the failing and passing test cases to encode the correct behavior as an input-output constraint for every part of the code that could be faulty, and then it exploits a database of patches written by developers in the past encoded as SMT formulas. The tool searches in the database to find if there is a formula that allows to satisfy the constraints. If it is found, it is then applied in the source code of the program. The process is repeated for every part of the program that is considered faulty.

Nopol [131] is able to create a patch for faults that are related to a wrong condition used in a if-statement or in a loop. The repair process is characterized by three main phases: 1) the angelic fix localization is used to find the values that the wrong condition should have to pass the test cases, 2) the runtime trace collection to collect the variables and their actual values, and 3) the encoding of these data into an instance of a Satisfiability Modulo Theory problem. If a solution to this problem is found, this represents the patch and it is applied to the program.

A similar approach to Nopol is Infinitel [55], that was designed to address problems of infinite loops. The technique determines the number of times that a loop should be executed to make the program pass all the test cases, and this number is called *angelic record*. The approach works in this way: it forces the condition of the loop to be evaluated true until the number of iterations is smaller than the angelic record, and to be evaluated false when they are equal. Once the constraints are found, it tries to solve the fault encoded as an instance of SMT problem.

2.1.4 Learning-based Repair Techniques

The recent improvements in the area of machine learning, especially in deep learning, and the large number of patches available in the online public repositories (e.g., GitHub, and GitLab) allowed developers to implement program repair tools that can *learn* how to generate a patch.

Unlike the template-based techniques, where the templates are defined manually considering also patches implemented by developers in the past, learning-based techniques automatically learn the templates analyzing the previous patches without human intervention. Thus, learning-based techniques can every time be trained with a different dataset.

One of the first example is Prophet [69], that exploits the information about software revision changes in order to improve the likelihood to have a correct patch. Indeed, the idea is that it is possible to find patches for similar bugs done in the past by the developers, and thus it is possible to exploit them to learn how to create a patch that addresses a specific type of fault.

Another technique is the one proposed by Long et al. [68], that infers code transformation templates from the patches developed in the past. In particular, the approach determines the AST-to-AST transformation templates used to create a patch, and then it uses these templates for its repair process.

There are then approaches that exploit the machine learning to train models that are able to predict the repaired code for a given piece of faulty code, without relying on a test suite or constraint solvers [62].

For example, DeepFix [54] trains a neural network to create patches that fix compilation errors. A compiler is used as an oracle to validate the generated patches.

R-Hero [10] is a repair bot that applies the continual learning to determine bug fixing strategies from continuous streams of commits that characterize the builds on Continuous Integration systems, such as Travis CI. The idea is that feeding continuous integration build streams to continual learning techniques, it is possible to develop a model that can be used to create patches for different types of bugs, including the compilations errors.

2.2 The Problem of Overfitting Patches

Even though a program repair tool modifies a program until none of the available test cases fails or crashes, the resulting program is not necessarily correct. In fact, test suites cannot feasibly cover every expected behavior of a program, and thus programs modified by the patches generated by APR techniques may yet be incorrect [136]. It is possible to distinguish between test-suite-adequate, overfitting, and correct patches.

Definition 2.2.1 (Test-suite-adequate patch (also known as plausible patch))**.** Given a program and its test suite containing at least a failing test case, a test-suite-adequate patch is a patch that makes the program pass all the available test cases. Test-suite-adequate patches are not necessarily correct. For example, a patch that simply removes the faulty instruction may make a program pass all the test cases without producing a correct patch.

Definition 2.2.2 (Overfitting patch)**.** An overfitting patch is a test-suite-adequate patch that is not correct because it does not comply with the intended semantics of the program, and thus it does not satisfy the requirements of the application. There are two kinds of overfitting issues: 1) incomplete fixing and 2) regression introduction [135]. The first issue is related to patches that work for the inputs used in the available failing test cases, but they do not work for every possible input that exercises the fault. The second issue is related to patches that work for all inputs that exercise the fault, but they break some already correct behaviors of the program.

Definition 2.2.3 (Correct patch)**.** Given a program and its test suite containing at least a failing test case, a correct patch is a test-suite-adequate patch that satisfies the requirements of the application.

Since the programs available in the benchmarks used to evaluate the effectiveness of program repair tools do not have requirements to establish how the applications should behave, it is difficult for researchers to determine if a patch satisfies the requirements of the application. For this reason, the evaluation of the correctness of a patch is often approximated as the capability of generating a patch equivalent to the one produced by the developers, if available.

Definition 2.2.4 (Equivalent patch)**.** Given a human patch and a patch generated by an APR tool, these two patches are equivalent if they are equal (i.e., they implement the same changes) or if the semantic of the changes is the same.

When the human patch is not available, the evaluation process is often limited to test-suite-adequateness, that is, to simply checking if a patch passes the available test cases.

Unlike the process used to establish if a patch is test-suite-adequate, which can be automated by simply running the available test cases, the process to establish if a patch is correct cannot be automatized and it is time consuming. Indeed, to evaluate the correctness of a patch, it is necessary to manually analyze the patch, to understand the context of the program in which it is implemented.

However, only considering test suite adequateness does not allow to distinguish between correct and overfitting patches, undermining the evaluation process. Subsection 2.2.1 discusses both the studies that analyzed this problem, and the solutions to mitigate it.

2.2.1 Studies about Overfitting Patches

The work by Smith et al. [107] is the first one about overfitting patches [82]. Their analysis shows that the quality of the patches produced by APR tools is proportional to the coverage of the test suite used during the repair process. In particular, adding a white-box test suite generated automatically with KLEE[1] to increase the code coverage obtained with a black-box test suite written by humans based on the program specifications, the patches generated by GenProg are overfitting, because they pass only the 75% of the new test cases generated with KLEE, and thus many of the patches should be discarded. This aspect points out that the test suite associated with the programs under repair is often not good enough to be used as an oracle for patch verification.

Also the study conducted by Qi et al. [95] shows that most of the patches generated by APR techniques are not correct, due to the weaknesses of the available test suites. Moreover, the study shows that patches often remove functionalities, simply avoiding the execution of faulty code, rather than repairing the code.

Approaches to Mitigate the Overfitting Patches Problem

Several studies investigated how to alleviate the problem of overfitting patches.

Xiong et al. [128] propose to exploit the behavior similarity of test case executions as a way to reduce the number of incorrect patches generated by APR tools. The idea is based on two observations: 1) when a correct patch is applied, a passing test case should behave similarly as in the buggy program, while a failing test case should behave in a different way; 2) when two test cases have similar executions, there is more likelihood to have the same test results. The similarity is measured considering the *complete-path spectrum* [35], that represents the sequence of executed statement IDs during a program execution. Their results are encouraging since the approach prevented the generation of 56.3% incorrect patches, without discarding any of the correct patches.

The analysis of Yu et al. [135] propose an approach to mitigate overfitting patches by using automatically generated test cases. The idea is that the repair constraints extracted using the test suite are not strong enough to completely express the intended semantics of a program, and thus augmenting the initial test suite with more automatically generated tests would allow to generate more correct patches. They demonstrated that their approach can reduce the issue related to the introduction of regression problems, but it has a minimal impact on reducing the number of incomplete patches. This is due to the fact that the test cases generated by their approach are not likely to be able to generate additional repair constraints for input points in the set of buggy inputs, and thus the overfitting patches that are incomplete patches cannot be detected with an high success rate.

[1]`https://klee.github.io`

Another study [17] uses the natural language processing on the source code in order to establish the reliability of patches generated by APR repair tools. The assumption is that a correct program is more similar to the original one compared to other patches that apply more changes in the program. The researchers found that the similarity is associated with the understandability, and when a patch is more understandable, the likelihood that a patch is correct is higher. To measure the source code similarity, they used Doc2vec [57] and Bert [22], that are two state of the art techniques used in the field of natural language processing. In the presented study, this approach successfully filtered out 16 out of the 35 overfitting patches under analysis.

Wang et al. [117] evaluated the effectiveness of the patch correctness assessment techniques comparing them on a dataset of 902 patches generated by 21 APR techniques. In particular, they considered 9 different techniques and 3 heuristics based on static code features. The nine techniques considered can be divided in two groups: 1) dynamic techniques that require an oracle (Evosuite [29], Randoop [88], DiffTGen [126], and Daikon [27]), and 2) dynamic techniques that do not require an oracle (PATCH-SIM [128], E-PATCH-SIM [117], R-Opad [134], E-Opad [134], and Anti-patterns [112]). The three heuristics are the ones used in three APR techniques and they do not require an oracle to work (ssFix [127], CapGen [121], and S3 [58]). In their study, they measured the precision and recall of the techniques, considering as true positive the overfitting patches identified as overfitting, false positive the correct patches identified as overfitting, false negative the overfitting patches identified as correct, and true negative the correct patches identified as correct. The result show that most of these techniques label correct patches as overfitting. This is due to different reasons that are related to the test cases generated by the tools. For example, in the case of Evosuite, the test cases generated by the tool broke the program preconditions, causing a wrong classification of the patches. In the case of DiffTGen, the problem is related to the fact that the test cases generated automatically are too strict (e.g., the test case checks not only if the program throws an exception, but also the message of the exception). Daikon uses extremely strict rules to compare identical invariants, checking also the line numbers of the exit points of a function, and this aspect causes the wrong classification of the patches. Finally, in other cases (PATCH-SIM and E-PATCH-SIM), the problem is related to the complexity of the original test suite, that leads to a wrong classification of the generated tests, and consequently also the patches are labeled in wrong way. Moreover, heuristics based on static code features allow to get higher level of recall, but they are less precise since they generate a greater number of false positives. These approaches are indeed not designed as a standalone technique to identify overfitting patches directly, but they have been analyzed to compare them to the standard techniques.

2.2.2 Challenge of Overfitting Patches in Automatic Program Repair

Although there are many different techniques in the field of APR, none of them is free of the problem of overfitting patches. As explained in Subsection 2.2.1, current studies explain the problem of overfitting with test suites that are often too weak, and thus they do not cover the programs well enough.

A family of overfitting patches is represented by code-removal patches, that are patches that drop functionalities of programs. Through the manual analysis of code-removal patches described in Chapter 3, it has been possible to discover that weak test suites are not the only reason that allows the generation of an overfitting patch. Indeed, there are other reasons, such as errors in the test cases, and rottening test cases (i.e., test cases that are executed only under some conditions).

2.3 Fault Localization

One of the main steps of APR techniques is the fault localization, which aims to automatically find the locations to apply the patches.

Spectrum-Based Fault Localization(SBFL), which uses test coverage information, is probably the most widely used fault localization approach [66, 30], since it is easy to apply and scalable to large programs [124]. A particular case, here defined as *augmented spectrum based* fault localization is represented by the techniques that enrich the information provided by SBFL techniques exploiting different sources such as the stack trace in the case of failures that generate an exception. Finally, a limited number of techniques use also information retrieval-based approach, which uses information contained in bug reports to find the locations that are likely to be faulty.

These techniques may identify the possible locations of the faults at different granularity levels, ranging from a file to a method or a line of code [66].

2.3.1 Spectrum-based Fault Localization

Spectrum-based techniques use the concept of path *spectrum*, that is a characterization of a program's execution based on a certain input. Different executions of the same program generate different path spectra, that thus correspond to different program behaviors [97]. Through the concept of program spectrum, it is possible for example to know how many times each statement of a program is executed [96]. Using the spectrum information for the fault localization, it is possible to identify which are the parts of the program involved with the failure so as to narrow the search space of the possible faulty locations. The idea is that program elements executed by many failing test cases and few passing test cases have more likelihood to be faulty. On the other hand, program elements executed by many passing test cases and few or no failing test cases have less likelihood to be faulty [30].

APR techniques exploited four main SBFL algorithms so far: Tarantula [43], Ochiai [2], Jaccard [15] and the algorithm defined in GenProg [61]. The next subsections describe the formulas used by each technique to localize the likely fault location, based on the spectra collected from passing and failing test cases. Here, we introduce the notation shared among the presented approaches.

In particular, to compute the suspiciousness of a statement s, it is used the following notation: *failed(s)* indicates the number of failed test cases in which the statement s has been executed one or more times, *passed(s)* indicates the number of passed test cases in which the statement s has been executed one or more times, *totalFailed* indicates the total number of failed test cases, *totalPassed* indicates the total number of passed test cases, and *execute(s)* indicates the number of test cases in which the statement s has been executed one or more times. For all the presented formula, the higher the suspiciousness of a statement is, the more likely the statement is faulty.

Tarantula

Tarantula computes the suspiciousness of each statement through the Equation 2.1.

$$suspiciousness(s) = \frac{\frac{failed(s)}{totalFailed}}{\frac{passed(s)}{totalPassed} + \frac{failed(s)}{totalFailed}} \tag{2.1}$$

The suspiciousness of a statement s can vary from 0 to 1. If any of the denominators evaluate to zero, the suspiciousness is computed as 0. The higher is the number, the more is the likelihood that the statement s is faulty. Thus, 0 is associated to the statements with less suspiciousness to be faulty, and 1 is associated with the statements with most suspiciousness to be faulty.

The idea behind Tarantula is that the statements that are executed mainly by failed test cases have more likelihood to be faulty compared to the ones executed mainly by passed test cases. Moreover, the formula allows some tolerance for the faults that are occasionally executed by passed test cases. According to the inventors of the formula, this aspect often improves the effectiveness of fault localization [43].

Ochiai

The Ochiai formula comes from molecular biology [87], and then Abreu et al. were the first ones to evaluate it in the field of fault localization [2].

Ochiai computes the suspiciousness of a statement s as the ratio between the number of failed test cases that execute that statement and the square root of the product between the total number of failed test cases and the sum of the failed and passed test cases that execute the statement s. More formally, Ochiai computes the suspiciousness of a statement s using the Equation 2.2.

$$suspiciousness(s) = \frac{failed(s)}{\sqrt{totalFailed \times (failed(s) + passed(s))}} \tag{2.2}$$

The suspiciousness value can vary from 0 to 1, and also in this case 0 is associated with the statements that are less likely to be faulty, while 1 is associated with the statements that are more likely to be faulty. Ochiai is often reported as one of the most effective technique in finding the root causes of faults [138, 125, 109].

Jaccard

To assign the suspiciousness score to a statement s, the Jaccard formula computes the corresponding value as the ratio between the intersection of the statements that execute s and the failed test cases (the numerator of the Equation 2.3) and the union between the statements that execute s and the failed test cases (the denominator of Equation 2.3).

$$suspiciousness(s) = \frac{failed(s)}{execute(s) + totalFailed - failed(s)} \tag{2.3}$$

The suspiciousness value can vary between 0 and 1, and like Ochiai, the higher is the value, and the higher is the likelihood that a statement is faulty.

GenProg Fault Localization Strategy

Unlike the other techniques, the approach used by GenProg to compute the suspiciousness of a statement s consists in assigning one of three different possible values, as shown in Equation 2.4.

$$suspiciousness(s) = \begin{cases} 0 & \text{if } failed(s) = 0 \\ 1.0 & \text{if } passed(s) = 0 \land failed(s) > 0 \\ 0.1 & \text{otherwise} \end{cases} \tag{2.4}$$

The value 0 is assigned to every statement that is not executed by any failed test case, the value 1 is assigned to each statement that is not executed by any passed test case and if there is at least one failed test case that executes that statement, while the value 0.1 is assigned to the statements that are executed by both failed and passed test cases.

In this way, the formula associates the biggest weight (1.0) with the statements that are executed only by failed test cases, because it is more likely that statements never involved in the executions of passed test cases are faulty. On the other hand, statements that are executed only by passed test cases should be correct and it is less likely that are part of the failure reason, and this why the formula assigns to them the weight 0. The value 0.1 associated with the statements executed by both passed and failed test cases allows to consider them for the generation of the ranking of suspicious statements, but with a lower score compared to the ones executed by only failed test cases.

2.3.2 Augmented Spectrum-based Fault Localization

To increase the likelihood to generate a correct patch, there are some tools that exploit additional information and not only use the suspiciousness ranking provided by SBFL techniques [130].

In some particular cases, program repair techniques can exploit more information to improve the ranking returned by spectrum based techniques. For example, HDRepair [59] assumes that the faulty methods are known, and only considers the lines of code inside these methods during the fault localization step. ssFix [127] gives priority to the statements contained in the stack trace of programs that crash. SimFix [42] exploits the concept of *test case purification* that consists in replacing every failing test case with k assertions by k single-assertion test cases and in removing the irrelevant statements for the failed assertion in each of them, with the goal to use these transformed tests to refine the ranking provided by Ochiai. ACS [129] uses the technique of *predicate switching* that consists in simulating the state changes by changing the branch predicate outcomes at runtime, to identify the most suspicious lines considering bugs related to faulty conditional statements [66]. The idea is to repeat the executions of the program on the failing input and switch the conditional branch outcomes until the program produces the correct output, so as to identify which are the conditional statements that are responsible of the fault.

There are then techniques that only consider a subset of the statements reported by Ochiai, such as SketchFix [37] that considers only the top 50 most suspicious statements, and Elixir [99] that considers only the top 200 most suspicious locations.

Finally, Guo et al. [33] propose to combine the dynamic slicing with spectrum-based fault localization, showing that it is possible to increase the performance of APR tools. Indeed, the results show that error propagation involves one single class for the majority of the faulty programs, and thus fault localization approaches should focus on the statements in the failure class reported in the bug report.

These approaches exploit specific assumptions that might be valid in certain contexts but that hardly generalize to every context.

There are also approaches that instead of exploiting only control-flow spectra, use also data-flow analysis. Data-flow analysis is related to the analysis of dynamic interactions between a memory definition *(def)* and subsequent *uses* of that definition during the program execution [105]. For example, Ribeiro et al. propose Jaguar [98], a tool to assist developers visually during the debugging, in which the suspiciousness ranking of *def-use associations* guides the search for the buggy program locations. In particular, Jaguar supports both control-flow and data-flow analysis, and it computes the suspiciousness score of each element according to a chosen ranking metric, such as Ochiai and Tarantula. Santelices et al. [100] propose a technique that first uses Tarantula to assign a suspiciousness score to every def-use association, then each of them is mapped to statements following three rules: 1) a def-use pair is associated with the *definition* statement; 2) a def-use pair is also associated with all statements that precede the *definition* statement in the same basic block; 3) an unmapped statement is associated with all def-use pairs whose uses are located in that statement. Then, the suspiciousness score of each statement is computed considering the highest score of all def-uses associated with that statement.

2.3.3 Information Retrieval-based Fault Localization

iFixR is a program repair tool that replaces the standard SBFL with information retrieval-based fault localization computed on the bug reports [52]. Indeed, the idea is to leverage the potential similarity between the terms contained in a bug report and the source code in order to identify which are the statements that are most likely faulty. Starting from the most suspicious files extracted with D&C [51], a tool that implements the Information Retrieval-based fault localization approach, the technique parses these files to get only the statements that according to the analysis of Liu et al. [65] are considered more error prone. These statements are if statements, expression statements, field declarations, return statements and variable declaration statements. Based on the experiment that the authors conducted, the results of Information Retrieval-based fault localization and spectrum based fault localization are comparable.

R2Fix [64] is another technique that exploits the information retrieval approach to localize the faults and it exploits free-form bug reports. The tool analyzes the bug report in order to extract the information about the bug type, and then it generates a patch based on that information.

Other approaches combine multiple fault localization techniques. For example, Motwani et al. [84] propose to use both spectrum based and Information Retrieval approaches. Based on their experiment that involved 818 real world defects, the unified approach is able to localize more bugs and rank better the suspicious statements compared to the single techniques.

Although promising, these approaches can be applied only in presence of bug reports filed by either users or developers.

2.3.4 Challenges of Fault Localization in Automatic Program Repair

In the field of APR, there are several studies about the influence of fault localization on the effectiveness of patch generation [120, 33, 133, 66]. Indeed, if localization is unable to identify the correct statements to be fixed, the APR tool cannot create a patch [66]. In the following, we discuss three main challenges that affect fault localization used in the context of APR.

Weak Discriminating Power A relevant challenge is about the low discriminating power of the techniques. Even though techniques give higher suspiciousness scores to the faulty statements, they tend to also assign high scores to statements that are not faulty. This aspect increases the risk that APR tools change also these statements, producing many variants of the original program that are not correct [133]. The study by Liu et al. shows that Ochiai, which is the most used fault localization algorithm in APR, ranks the faulty statements at the top position for only 11% of the investigated faults, and in one of the top ten positions for only 35% of the investigated faults [66]. The study by Assiri and Bieman reports a mean position of the faulty statement in the ranking returned by Ochiai equals to 26.1 [6]. Finally, Pearson et al. provides extensive empirical evidence of the difficulty of SBFL techniques in identifying patch locations with real faults [91]. The results of the analysis conducted by Wen et al. show that the accuracy of fault space, that is the ranked list of the suspicious statements affects the effectiveness and the efficiency of search based APR techniques [120]. Considering GenProg, the results indicate that it is likely to generate more patches in a faster way when the mean average precision of the fault space has an accuracy over 0.9.

Size of the Rank A second challenge of fault localization techniques is related to the number of statements that are reported. For big programs, the number of top ranked statements can be more than one hundred. Large ranks increase the likelihood that APR techniques try mutating many potentially irrelevant code location, dramatically increasing the time of the repair process [133].

To mitigate this problem, xJiang et al. [41] performed a manual analysis of 50 defects from Defects4J, and proposed to incorporate richer dynamic information about test failure in the ranking algorithm in order to exclude locations associated with methods that are unlikely to be faulty, such as methods of library function or methods that are just wrapper of others. In line with the work described in this thesis, they also suggest to try to introduce a way to identify the statements that introduce undesirable value changes in a test execution, since these statements are the ones responsible of the fault.

Nothing more than locations Another challenge is related to the weak integration between the fault localization and the patch generation steps. Indeed, fault localization techniques suggest a ranked list of statements, but there is no information about how these locations may relate to the failures and how they should be changed. For example, if a suspicious location is related to the initialization of an array, giving this extra information to the repair step may allow program repair techniques to create certain types of patches targeting that specific problem, that is the initialization of the array. This aspect allows to guide the repair process, giving the possibility to limit the search space and to create patches based the guessed fault. How to generate this additional piece of information is also studied in this thesis.

In this context, considering these challenges, Chapter 4 describes a new exception-driven fault localization technique that, exploiting the semantic of the exceptions, provides some specific locations related to the exception, enriched with information about the guessed fault. Program repair techniques could exploit this extra information to apply only specific types of change based on the guess.

Chapter 3

Effectiveness of Code-removal Patches

This chapter presents the study about the factors that may influence the generation of code-removal patches and the information that can be derived from them. The analysis results in the definition of a comprehensive taxonomy of code-removal patches that can be exploited to better understand the current limitations of program repair techniques. The analysis also includes a comparison of human patches and code-removal patches, demonstrating the possibility to extract valuable information from the analyzed patches.

3.1 Code-removal Patches

As described in Chapter 2, one of the limitations of APR techniques is related to the generation of overfitting patches. Indeed, current repair techniques rely on the test cases to evaluate the candidate solutions, but this is a too weak validation method, and it causes the generation of patches that are not correct, even though they pass all the test cases. The overfitting patches represent a serious problem because they apparently fix the bug, but actually they can introduce new bugs not revealed by the test cases, or they can also drop the functionality so that the previous failing/crashing test cases are no longer executed. In this last case, the buggy functionality is not fixed, but it is simply removed from the program, and so the patch is meaningless. Analyzing this type of patches in order to understand which are the reasons that make possible their creation is important to improve the current repair techniques and also to find a way to exploit them in order to automatically improve the test suite associated with the programs. Code-removal patches, generated by the Kali system [95] (presented in Subsection 2.1.2) are an example of overfitting patches. There are multiple implementations of the Kali strategy. In particular, Astor is an automatic software repair framework for Java programs that implements different repair approaches, including jKali, a Java implementation of Kali [95]. In some cases, when it is correct to delete some part of the code to fix the bug, these patches are correct, but they are overfitting most of the time.

Definition 3.1.1 (Code-removal Patch)**.** A code-removal patch is a patch that simply removes functionality, by deleting, or skipping code. This latter case can be done through the replacement of a condition in order to force the execution of a specific branch, or through the addition of a `return` statement in a function body. Despite functionality removal, a code-removal patch may change a program making a full test suite to pass [95].

Listing 2 shows an example of a code-removal patch generated by Kali for the bug php-309892-309910, related to the PHP's standard library function `substr_compare`. In this case, the code-removal patch changes the `if` condition adding the instruction `&& !(1)`, thus the body of the `if` statement is skipped, because the condition is always evaluated to false. This patch is semantically equivalent to the patch implemented by the developers, which entirely removes the `if` statement [13].

```
-   if (len > s1_len - offset) {
+   if (len > s1_len - offset && !(1)) {
        len = s1_len - offset;
    }
```

LISTING 2: Example of code-removal patch generated by Kali for the bug php-309892-309910.

Interestingly, recent empirical studies show that Kali is effective in finding test-suite-adequate patches in large and complex systems [95, 69]. The intuition behind the study presented in this chapter is that the very presence of code-removal patches carry some meaning, that can be exploited to fix code.

In particular, the study investigates the reasons and the scenarios that make the generation of code-removal patches possible, and the relation of code-removal patches with the human-written patches. In addition, the study investigates if code-removal patches carry valuable information for developers, even when they are incorrect.

The proposed analysis results in a comprehensive taxonomy of code-removal patches that both improves the level of understanding of program repair techniques and that can be exploited to support program debugging tasks performed by the developers.

3.2 Experimental Methodology

This section describes the experimental procedure and discusses the design choices relevant to the study.

3.2.1 Goals & Research Questions

The goal of this study is to understand the nature of test-suite-adequate code-removal patches and to investigate how these patches relate to human patches, with a thorough qualitative study. To our knowledge, this is novel in program repair research, nobody has ever studied this important point.

In particular, the study aims to answer to the following research questions:

RQ1 What is the relation between assertion failures and the generation of test-suite-adequate code-removal patches?

This research question investigates the ability of jKali to generate code-removal patches for tests that fail due to the violation of an assertion.

RQ2 What is the relation between crashing tests and the generation of test-suite-adequate code-removal patches?

This research question investigates the ability of jKali to generate code-removal patches for the crashing faults revealed by crashing test cases.

RQ3 To what extent can code-removal patches, even if incorrect, give valuable information to developers to find weaknesses in test suites?

The goal of this research question is to study if a code-removal patch can give valuable information about the cause of the problem, while revealing weaknesses in test suites, regardless of its correctness.

RQ4 How do developers fix the failed builds associated with a test-suite-adequate code-removal patch?

The goal of this research question is to study if developers fix bugs according to some patterns when the bug can be also addressed with a code-removal patch. The study also investigate the semantic and syntactic similarities between the patch produced by developers and the patch produced by jKali.

The experimental methodology is organized in three main phases: the collection of build failures, presented in Subsection 3.2.2; the analysis of the collected build failures, to identify the ones amenable to code-removal patches, presented in Subsection 3.2.3; and the analysis of the code-removal patches and their comparison to developers' patches, presented in Subsection 3.2.4.

3.2.2 Data Collection

The first phase of the study consists in collecting the build failures, and the related artifacts, necessary to answer to RQs 1-4. As source of build failures, the study considers the Repairnator-Experiments repository.

Definition 3.2.1 (Repairnator-Experiments)**.** Repairnator-Experiments is an open science repository[1] that contains the metadata information of the Travis CI builds that Repairnator tried to repair [115]. It hosts 14,132 failed builds (collected in the period February 2017 - September 2018) for 1,609 Java open-source projects hosted on GitHub. It provides detailed information about the builds, such as the event that triggered the build, the number of failing and crashing test cases, and the type of failures. Moreover, for every build, Repairnator-Experiments stores the source code associated with the failing commit.

Definition 3.2.2 (Repairnator)**.** Repairnator is a software engineering bot that monitors program bugs discovered during Continuous Integration, and tries to fix them automatically [83]. Repairnator implements multiple repair strategies, including the generation of code-removal patches using jKali.

For each failed build, the following artifacts are considered:

- The *failure-related artifacts*.
- The *code-removal patches* generated for the failed builds.
- The *human patches* associated with the failed builds that also have a code-removal patch.

All these data have been saved on a GitHub repository[2] for the sake of scientific reproducibility. The description of how these artifacts were obtained is provided below.

Failure-related artifacts. The builds relevant to the study are all the builds stored in the Repairnator-Experiments repository with either one failing or one crashing test case. In total, there are 2,381 builds with only one failing test case, and 1,724 builds with only one crashing test case. Overall 4,105 out of the 14,132 builds (29.05%) satisfy the selection criterion. For each failed build, from the Repairnator-Experiments repository, the following items have been collected: the source code versioning the commit that made the build fail, the metadata associated with the failed build (e.g., the build id), and finally the information about the failure (e.g., the name of the class and the test method that fails or crashes).

Code-removal patches. For every selected build, jKali was run to generate code-removal patches, if such patches exist. The jKali patches already present in the Repairnator-Experiments repository were not used because they have been generated with a previous version of jKali, which was affected by several bugs that are now fixed. These new runs ensured that a project code associated with a failed build is still executable. In fact, sometime there are problems with deletion of branches or dependencies that now make the execution of jKali impossible.

[1]`https://github.com/repairnator/repairnator-experiments`
[2]`https://github.com/repairnator/open-science-repairnator/`

Given the computational nature of the task, jKali was run with a timeout of 3 hours for every build. This 3 hours period comprises all phases of jKali, from downloading the source code from the repository to the execution of the repair attempts. The core repair loop of jKali itself is set to run for a maximum of 100 minutes. This setup is based on previous research [25][3]. Note that jKali is set to stop after finding the first test-suite-adequate patch, thus for every failing build there is at most one code-removal patch. This process produced 48 code-removal patches (27 patches for builds with a failing test case and 21 patches for builds with a crashing test case).

Human Patches. The data collection step finally includes the identification of the human patches associated with the builds for which jKali created a code-removal patch. To determine the corresponding human patch, automatic and manual analysis are performed: Travis CI API[4] is used to automatically determine the commits that may include the patch and the presence and appropriateness of the human patch is manually checked.

Using the Travis CI API, the first build 1) that follows the one under analysis, 2) that is on the same branch and 3) that is associated with a status `passed` is retrieved. Then, all the commits between the failed build under analysis and the first passed build are extracted.

If only one commit with a simple change is extracted, the patch is confirmed by directly applying this transformation to the failed build and running the test suite. If a commit includes multiple changes, the likely fixing changes are selected manually and they are applied to the code that fails the build. If the build passes all the available tests with the extracted change, this change is considered as the ground-truth human patch.

Sometimes, the set of commits contains too many different changes involving many parts of the program, and it is infeasible to precisely identify the ones that actually contribute to fixing the bug. Overall, this combination of automated and manual analysis has taken 3.5 hours per code-removal patch on average and 21 days in total.

3.2.3 Analysis of Failed Continuous Integration Builds

The second part of the study consists in the analysis of the collected builds, which is organized in the following two steps:

- A sanity check of the builds is performed.
- Essential information about the failure is retrieved.

Sanity Check of Builds. The goal of the first step is checking if the results obtained with jKali are the same as the ones reported in the Repairnator-Experiments repository. Indeed, even if a build failed in the past, there is no guarantee that the

[3]Repair tools demonstrated to require 13.5 minutes on average to generate a patch, so allocating 100 minutes is 7.4 times the average repair time.

[4]`https://docs.travis-ci.com/user/developer/`

same build fails with the same error after several months: this is due to changing external dependencies, closed third-party services that the application under test uses, flakiness[5] of the tests, etc. In addition, the selection criterion requires builds that include only one failing or crashing test case. To sum up, the sanity check ensures the following two conditions:

- the build process terminates correctly,
- the execution of the tests terminates with either a failing or a crashing test case

Thus, the 4,105 selected builds were downloaded and their test cases were executed, discarding the builds that did not pass the sanity check. At the time of the experiment in March 2020, this step results in 2,187 builds discarded for the following reasons:

- 635 builds had errors during the build phase (e.g., because some dependencies are not resolvable);
- 393 builds passed all the test cases, suggesting the presence of flaky tests causing the failure on the first place;
- 75 builds generated a timeout during the execution of the test cases;
- 1,084 builds had multiple failing and crashing test cases.

After the sanity check, the number of relevant builds for the study amounts to 1,918. In particular, 950 builds have only one failing test case, and 968 builds have only one crashing test case.

Retrieval of Information about the Failure. The second step consists of collecting qualitative and quantitative information about the failed builds. For the builds with one crashing test case only, the type of the exception responsible for the failure (e.g., NullPointerException) is collected. For the builds with a failing test case only, information about the type of the failure is collected. However, differently from crashing test cases, failures do not have an explicit type information assigned. Thus, failures were classified based on a manual analysis of the test and of the failed assertion in particular.

Table 3.1 shows the categories that were identified. The first column (*Failing Assertion Category*) specifies the category of the assertion failure, the second column (*Definition*) defines the category, and finally the third column (*Example of potentially failing test code for this reason*) contains a sample failure assertion extracted from the benchmark to exemplify the category.

Wrong Values generally represents all those cases in which there is a difference between the expected value and the one generated by the program under test. The

[5]As observed in a study conducted by Durieux et al. [24], 0.80% of the builds of their dataset, which contains 3,286,773 Travis CI builds, are flaky. In particular, 46.77% of the restarted builds (1.72% of the builds of their dataset) change their state from *failed* to *passed*. Thus, builds that fail due to flaky tests are not unusual.

TABLE 3.1: Classification of reasons for failing test cases.

Failing Assertion Category	Definition	Example of potentially failing test code for this reason

Wrong Values	The value generated by the program is not acceptable according to the expected value in the test assertion.	`CloseableHttpResponse response =`
		`    httpclient.execute(httpGet);`
		`assertEquals(200,`
		`    response.getStatusLine().getStatusCode());`

Mocking Verification Failure	The test execution does not pass a check performed on a mocked component.	`Processor<String> mockProc = mock(Processor.class);`
		`verify(mockProc, times(2)).process(eq("k1"));`

Exception Difference	The failure is caused by the generation of an exception of the wrong type or with the wrong message, or by a missing exception.	`@Test(expected = ParseException.class)`
		`public void whenGivenBadScnlThrowHelpfulExcept() {`
		`    parser.parse("not a SCNL");`
		`}`

Timeout	The failure is caused by the program not generating an output in the maximum allowed time.	`TestObserver<String> test = rxResponse.test();`
		`test.awaitTerminalEvent(1, TimeUnit.SECONDS);`

Environment Misconfiguration	The failure is caused by an incorrect execution environment (e.g., an environment variable is not set).	`assertNotNull("ensure ${env.CI_OPT_MVN_CENTRAL_USER}`
		`    and ${env.CI_OPT_MVN_CENTRAL_PASS} is set.",`
		`    plainText);`

code snippet shows an example of a test case whose aim is to verify if the status code of the HTTP response is 200.

Mocking Verification Failure represents a failure reported by a Mock framework, for instance because a specific method is not called the expected number of times. The code snippet shows the case of a failure reported by Mockito because the method `process` is called a number of times different than 2 with the argument `k1`.

Exception Difference represents failures caused by a difference between the observed exception and the expected exception. The code snippet shows the case of a test case that fails because the expected exception of type `ParseException` is not generated.

Timeout represents tests that fail due to the timeout of an operation. The code snippet shows an example of a test case that fails because the `Subscriber` does not receive a notification within a second from the time `Observable` finished its job.

Finally, *Environment Misconfiguration* represents failures caused by problems in the testing environment. The software environment includes every entity external to the program, such as configuration files, system variables, external programs. The code snippet shows a test case that fails because the username and password associated with the Maven environment have not been properly set up.

3.2.4 Analysis of Human Patches and Automated Code-removal Patches

The third part of the methodology consists of a quantitative and qualitative analysis of both the generated code-removal patches and the ground truth human patches. The rest of this section presents the analysis and the identified categories.

Classification of Code-removal Patches. Code-removal patches are analyzed to first determine their correctness. To this end, similarly to previous studies [74, 95], a patch is considered to be correct if it is either identical or semantically equivalent to the corresponding human patch.

In Listing 3, the case of a code-removal patch generated by jKali for the failing Travis CI build with id 322406277[6] associated with the project pac4j is exemplified. This patch is semantically equivalent to the human patch, and is thus considered correct. The code-removal patch changes the `if` statement using `false` as condition, which forces the execution of the `else` branch, and consequently forcing the method to always return `null`. The human patch removes the overriden method shown in Listing 3[7]. The program without the overridden method has the same behavior as the program with the code-removal patch. Thus, even though the code-removal patch does not entirely delete the overriden method `internalConverter(Object)`, it generates a program with the same behavior of the one that includes the human patch.

```
    @java.lang.Override
    protected String internalConvert(final Object attribute) {
-       if (null != attribute) {
+       if (false) {
            return attribute.toString();
        } else {
            return null;
```

LISTING 3: Example of code-removal patch generated by jKali for failing Travis CI build 322406277.

If the code-removal patch is not correct, it is classified according to its nature. In particular, four different potential issues affecting the test cases that caused the acceptance of wrong code-removal patches were identified: Weak Test Suite, Buggy Test Case, Rottening Test, and Flaky Test. Table 3.2 summarizes these cases.

Well known in the literature, a *weak test suite* might be responsible of the acceptance of a wrong code-removal patch. This was confirmed by showing that assertions can be manually added to existing tests or that new test cases discarding the code-removal patch can be added.

[6] `https://travis-ci.org/github/pac4j/pac4j/builds/322406277`
[7] `https://github.com/pac4j/pac4j/pull/1076`

Sometime there are *buggy test cases*, that is, a wrong code-removal patch is not discarded because the test expects the wrong behavior from the program. This is revealed and confirmed by manual patches made to the test cases after the failure.

A *rottening test* is a test that contains assertions that are executed only based on some conditions [21]. If the code-removal patch changes the program in a way that the execution of the assertion is skipped, the test is now green (because the assertion is not executed) and the patch is accepted even if the program is still faulty. This is the first work showing that such test issues affect the acceptability of program repair patches.

Finally, the presence of *flaky tests* [72] may let jKali accept a patch. When a flaky test stops failing, it has actually no causal relation with the generated patch, and the patch is then wrongly assumed as being correct.

TABLE 3.2: Classification of code-removal patches. While the literature has focused on WT, reasons CP, BT, RT and FT have never been studied before.

	Category	Definition
	CP: Correct Patch	The code-removal patch is equal or semantically equivalent to the human patch.
Wrong Patch	WT: Weak Test Suite	The available test suite does not cover the program well enough, better assertions and better tests might be needed.
	BT: Buggy Test Case	The code-removal patch works due to a fault in a test case (e.g, the expected value in the test case is not correct).
	RT: Rottening Test	The code-removal patch disables the execution of the failing assertion, that is located in a control flow statement. This means that the code-removal patch affects the return values used in the expression of the control flow statement containing the failing assertion, thus avoiding its execution.
	FT: Flaky Test	A code-removal patch is accepted due to a flaky test that now passes.

Classification of Human Patches. In this study, for 32 out of 48 cases (66.67%) the human patch corresponding to a code-removal patch is found. For each of them, the patch is classified based on the type of changes implemented by the developers. The classification takes into account the size of the change (e.g., if changes are localized in a statement or in a method) and the target of the changes (e.g., if the changes target the program or the tests). Code-removal changes made by developers which match the type of changes produced by jKali were also looked for. Table 3.3 shows a summary of all the categories found by analyzing the patches.

TABLE 3.3: Classification of human patches when a code-removal patch exists.

Patch Type	Category	Definition
Fix in Test	Fix Test Code	The patch fixes the logic of one or more test cases to properly reflect the expected behavior.
	Fix Test Data	The patch fixes the data used in one or more test cases.
Statement-Level Change	Change Condition	The patch modifies a condition used in the program, for example in a `if-statement` or in a cycle.
	Add if-else Statement	The patch adds a new `if-else` statement to conditionally execute part of the code.
Method-Level Change	Change Method Implementation	This patch modifies multiple statements inside a same method.
	Override Method	This patch introduces an method that overrides another method present in the program.
Code Removal	Remove Variable Assignment	This patch removes an assignment statement from the program.
	Remove Variable Annotation	This patch removes an annotation from the program.
	Revert	The patch reverts to a previous commit.
Not Available	No Change	This patch includes no changes, or if they are present, they are not related to the failure, typically because the build failure is due to the presence of flaky tests.
	Not Found	It is not possible to determine the changes that make the patch, for instance because they are mixed with many other changes not related to the removed fault.

The first two categories, *Fix Test Code* and *Fix Test Data*, correspond to patches implemented by changing the code of the test, while distinguishing between changes to the logic of the tests and changes to the data used in the tests. A number of categories capture the case of actual modifications implemented in the code of the faulty program (excluding code-removal only patches). These changes might be at the level of the *individual statements* or at the level of the *methods*. Changes to individual statements involved either conditions or if-else statements. Although other types of changes to individual statements are possible, patterns for which no code-removal patch exists in the dataset of this study are not listed. Changes to methods involved either a method or the addition of an overriden version of a method.

Interestingly, a number of human patches that consist of *Code Removal* operations have been found. Cases in which the developers removed assignments, annotations or revert a code change are reported.

Finally, sometimes the human patch is not available, either because the failed build has been intentionally not fixed (e.g., because the failure was caused by a flaky test) or because the procedure used was not able to uniquely identify the human patch, as already described.

3.2.5 Summary

In this section, a novel methodology to analyze program repair patches has been presented. In particular, the taxonomy of failures (Table 3.1, the classification of code-removal patches based on the causes that make them work (Table 3.2), and the categorization of ground-truth human patches (Table 3.3) is novel. This can provide a solid foundation for future studies of program repair patches.

3.3 Experimental Results

This section presents the results of the analysis and provides the answers to the research questions.

3.3.1 What is the relation between assertion failures and the generation of test-suite-adequate code-removal patches? (RQ1)

In this research question, the relation between the category of assertion failures and the proportion of generated code-removal patches is analyzed per the methodology of Section 3.2. Table 3.4 reports the information about the number of builds per category of assertion failures and the corresponding code-removal patches generated by jKali.

TABLE 3.4: Relation between the test failure categories and code-removal patches.

Failing Assertion Category	Occurrences	# Builds with Patch	% Builds
Wrong Values	843	21	2.49%
Exception Difference	39	3	7.69%
Mocking Verification Failure	20	2	10.00%
Timeout	46	1	2.17%
Environment Misconfiguration	2	0	0%
Total	950	27	2.84%

The first column (*Failing Assertion Category*) lists the different categories of assertion failures, the second column (*Occurrences*) shows the number of builds that have a failing test case for each category of assertion failure, the third column (*# Builds with Patch*) indicates the number of builds that have a code-removal patch generated by jKali, and the fourth column (*% Builds*) indicates the percentage of builds with a code-removal for every failing assertion category and over all 950 builds. The data are presented in descending order by the number of patched builds.

Analysis of the Results

The most frequent category of assertion failure is *Wrong Values* with 843 occurrences, while the least frequent category is *Environment Misconfiguration* with 2 occurrences.

jKali generates a patch for all failure categories, with the exception of *Environment Misconfiguration*. This is due to both the few occurrences in this category, but also to the missing capability of changing the environment configuration files in jKali. In fact, jKali is designed to create patches that change the source code of the target program, and cannot make any change to the environment.

Wrong Values is the category of assertion failure with the highest number of occurrences (843) and the highest number of code-removal patches (21). The high number of patches is only due to the high number of failing builds in that category. The assertion failures with the highest percentage of code-removal patches are *Mocking Verification Failure* (10.00%) and *Exception Difference* (7.69%). While this percentage is high, it is still a rare event and the absolute number of patches is still low (2 for *Mocking Verification Failure* and 3 for *Exception Difference*). Due to this event rarity, it is not possible to make strong claims that those failures types are more amenable to code-removal patches.

To study if the generation of code-removal patches is dependent on the category of the assertion failure, Fisher's exact test is applied on the number of patched builds per assertion failure category. The null hypothesis of the test is that *the number of patched builds is independent of the assertion failure category*. The p-value is 0.07852, that is greater than the significance level α set to 0.05. This means that the null hypothesis cannot be rejected, and the capability to generate code-removal patches is independent of the category of assertion failure.

Comparison of the Results with Previous Studies

Another interesting aspect is that the proportion of generated code-removal patches is significantly lower than in previous studies. Indeed, in this case, the likelihood is 2.84%, while in the previous studies is 36.23% [69], 25.71% [95], and 9.28% [74]. The best explanation is that it is due to the small size of the datasets used in former experiments, which consist of 69 cases for [69], 105 cases for [95], and 224 cases for [74]. A second explanation is that those previous studies did not sample over builds but over commits. A third explanation is that the benchmarks of those studies used some kind of selection, which results in a biased sampling.

Yet, the result is aligned with Durieux et al. [25], in which the proportion of patches generated by jKali on the bugs of Bears is about 3.0%. Unlike the other benchmarks, Bears is a benchmark which uses CI builds to identify buggy and patched program version candidate, and it contains bugs associated with more different programs (72 projects) compared to the other ones (8 for ManyBugs benchmark [60], and 5 for Defects4J benchmark [44]). Both Bears and this study sample builds, which explains the strong consistency.

What is the relation between assertion failures and the generation of test-suite-adequate code-removal patches? (RQ1) jKali has been able to create a patch for 27 out of 950 (2.84%) builds having only one failing test case. One patch for every category of assertion failure was obtained, with the exception of *Environment Misconfiguration*, which is out of scope for current generators of code-removal patches. The results show that the generation of a code-removal patch is independent of the category of assertion failure. The analysis suggests that former studies tended to over-estimate the prevalence of code-removal patches, because of the selection criteria considered. The results are useful for program repair researchers, they give a better understanding of the somewhat limited repair capability of code-removal patches.

3.3.2 What is the relation between crashing tests and the generation of test-suite-adequate code-removal patches? (RQ2)

In this research question, the relation between the type of exceptions and the proportion of generated code-removal patches for crashing tests is analyzed per the methodology of Section 3.2. Table 3.5 shows the details about the number of available builds, divided per exception type, and the corresponding number and percentage of patches produced by jKali.

The first column (*Exception Type*) indicates the name of the exception, the second column (*Occurrences*) reports the number of builds having a crashing test case with the specific type of exception, the third column (*# Builds with Patch*) indicates the number of builds that have a code-removal patch generated by jKali, while the fourth column (*% Builds*) reports the percentage of code-removal patches for every type of exceptions and over all 968 builds. The data are reported in descending order by the number of patched builds including only the exceptions for which there is at least one code-removal patch. In total, jKali is able to create a patch for 21 out of 968 builds with a crashing test case, with a success rate of 2.17%.

Comparison of the Results between Builds with Failing and Crashing Test Cases

The success rate reported for crashing failures is in line with the success rate reported for builds with failing test cases (2.84%). To study if the nature of the failure, produced by either a crashing or a failing test case, has an impact on the capability

TABLE 3.5: Relation between the types of crashing exceptions and code-removal patches.

Exception Type	Occurrences	# Builds with Patch	% Builds
NullPointerException	124	7	5.65%
Exception	66	3	4.55%
OutOfMemoryError	5	2	40.00%
ClassCastException	7	2	28.57%
FileNotFoundException	27	1	3.70%
IllegalArgumentException	54	1	1.85%
IllegalStateException	241	1	0.41%
javax..PersistenceException	2	1	50.00%
rocketmq..MQClientException	1	1	100.00%
RuntimeException	62	1	1.61%
ConnectorStartFailedException	1	1	100.00%
Other exceptions	378	-	-
Total	968	21	2.17%

to produce a code-removal patch, the Fisher's exact test was applied considering the following null hypothesis: *the number of patched builds is independent from the type of test case that reveals the failure (either a failing or a crashing test case)*. The p-value is equal to 0.3833, that is higher than the significance level α set to 0.05. The null hypothesis is thus rejected, indicating that the generation of code-removal patches is independent from the type of test case that reveals the failure.

Analysis of the Results

The most frequent type of exception is `IllegalStateException`[8] with 241 occurrences, but only one build has a code-removal patch. This type of exception occurs when a method has been invoked at an inappropriate time, thus a code-removal patch has a low likelihood to make pass a crashing test. Indeed, a code-removal patch can remove the wrong method call, but to avoid that the exception is thrown, it is usually also necessary to replace the removed call with the right piece of code (e.g., the invocation of another method), in order to create a legal program state. However, this is outside the scope of code-removal patches, that can only remove, but not add code.

A larger number of code-removal patches for `NullPointerException` (7) and the generic `Exception`[9] (3) is reported. This result can be explained by the fact that these exception types are prevailing in the benchmark. Furthermore, they were manually analyzed. It is observed that jKali successfully patches programs producing a

[8]`https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/IllegalStateException.html`

[9]`https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/Exception.html`

NullPointerException by removing the usage of the object that is null. Regarding the generic exception java.lang.Exception, the 3 code-removal patches are related to timeout errors and they remove the piece of code that causes the timeout. For example, the code-removal patch of the Travis CI build for Apache Twill (id 356030973) removes the call to method java.net.InetAddress.getLoopbackAddress(), whose execution could generate a deadlock.

Some exceptions only sporadically present in the benchmark have been successfully patched, that is the case for MQClientException[10], ConnectorStartFailedException[11], and PersistenceException[12]. The number of builds with these types of exceptions is far too low to be able to generalize a finding. Finally, it is reported that OutOfMemoryError and ClassCastException have been patched with good frequency. The first one is an error that is thrown when there is insufficient memory for the program to work properly. The success rate of jKali is 40%. The second one is an exception that occurs when there is an instruction in the source code that tries to convert an object from one type to another, but they are incompatible (e.g., converting an Integer to a String). In this case, the success rate of jKali is 28.57%.

To confirm that the presence of code-removal patches does not depend on the exception type, the Fisher's exact test is used with the hypothesis: *the number of patched builds is independent of the crash category*. The p-value is equal to 0.3831, that is greater than the significance level α set to 0.05, and thus there is no evidence to reject the null hypothesis.

What is the relation between crashing tests and the generation of test-suite-adequate code-removal patches? (RQ2) jKali has been able to create a patch for 21 out of 968 builds having one crashing test case only (2.17%). The most repairable type of common exception is OutOfMemoryError (5 occurences), and NullPointerException is a prevalent exception with code-removal patches (7). The experiment shows that the generation of code-removal patches is independent on the exception type in a statistically significant manner. This experiment shows that IllegalStateException is a very common kind of exception, for which specific program repair tools are needed.

[10]https://rocketmq.apache.org/docs/quick-start/

[11]https://docs.spring.io/spring-boot/docs/1.5.7.RELEASE/api/org/springframework/boot/context/embedded/tomcat/ConnectorStartFailedException.html

[12]https://docs.oracle.com/javaee/7/api/javax/persistence/PersistenceException.html

3.3.3 To what extent can code-removal patches, even if incorrect, give valuable information to developers to find weaknesses in test suites? (RQ3)

In this research question it is manually analyzed why code-removal patches have been generated, and it is studied if these patches can provide valuable information about the cause of the build failure, also revealing weaknesses in test suites.

TABLE 3.6: Relation between the failing builds and code-removal patches.

Patch reason	# Builds - fail. test	# Builds - crash. test	Total
Correct	1	1	2
Weak Test Suite	11	14	25
Buggy Test Case	5	2	7
Rottening Test	4	2	6
Flaky Test	6	2	8
Total	27	21	48

Table 3.6 shows the different reasons that lead to the generation of a code-removal patch. The first column (*Patch reason*) lists the possible reasons, the second column (# Builds - fail. test) and the third column (# Builds - crash. test) report the corresponding number of builds with a failing and crashing test case, respectively. Finally the fourth column (*Total*) shows the total number of the patched builds for every patch reason.

Correct Patches

Only 2 out of 48 code-removal patches (4.17%) are correct. This confirms previous research showing that code-removal patches are mostly incorrect [95, 74].

The other 46 patches are all different manifestations of the inadequacy in the test suite.

Weak Test Suite

For 25 out of 46 patches (54.35%), the problem is a *Weak Test Suite* that does not sufficiently assert the behavior of the program under test. For example, build `400611810` generates an assertion error when comparing the expected status code with the actual one. As shown in Listing 4, the code-removal patch removes the instruction that adds the HTTP header, and in this way the resulting HTTP answer has the expected status code. The patch works because test[13] checks only if the status code is correct, without checking if the HTTP request contains the right header.

[13]`https://github.com/repairnator/repairnator-experiments-one-failing-test-case/blob/12171dada351fd2bfe999f8dd10cb0931829b5fb/src/test/java/com/http/TestRequest.java#L11`

```
--- /src/main/java/com/http/Request.java
+++ /src/main/java/com/http/Request.java
@@ -235,7 +235,6 @@
    header.put("Accept-Encoding", "gzip, deflate, br");
    header.put("Accept", "text/html,application/xhtml+xml,application/xml;
                          q=0.9,image/webp,image/apng,*/*;q=0.8");
    header.put("Connection", "Keep-Alive");
-   this.setHeader(header);
}

public java.lang.String getContent() {
```

LISTING 4: Example of code-removal patch generated by jKali for build 400611810.

Buggy Test Case

For 7 out of 46 patches (15.22%), the code-removal patch reveals a *Buggy Test Case*, that is, a faulty test case that allows the acceptance of an incorrect patch. For example, build `35121194` of the dataset of this study fails after the addition of a change that trims the output associated with the result of a command execution, but the test case is not updated to support this change. The code-removal patch works because it removes exactly the new instruction that trims the output (i.e., the patch undoes the change), and so the test case passes. To our knowledge, this is the first ever report of this phenomenon in the literature.

Rottening Test

It is observed that the acceptance of incorrect test-suite-adequate patches has been caused by *Rottening Test*s in 6 out of 46 cases (13.04%). In such cases, the failing or crashing test case's assertion is no longer executed after the application of a code-removal patch. This result confirms that change in the application code can have an effect on the test execution [75]. For example, build `38897112` generates a `ClassCastException` exception when a test case checks the value associated with a property of a JSON object under test. Since the check is executed only if the object has the property, and since the code-removal patch removes the instruction to set the property of the JSON object, the patch passes all tests because the assertion is not executed. To our knowledge, this is the first ever report of this phenomenon in the literature.

Flaky Test

Finally, another interesting category is *Flaky Test* with 8 out of 46 incorrect test-suite-adequate patches (17.39%). In these cases, the patch is accepted because of a *Flaky Test*, i.e. the patch is not related to the test pass. These code-removal patches make irrelevant changes that do not modify the logic of the program, (e.g., printing of log information, as observed for build `40344741` on Travis CI), but end up being accepted as side effect of the intermittent failures generated by flaky tests (e.g., if the flaky test does not fail after an irrelevant change is introduced in the program, the change is reported as an acceptable patch). An easy way to mitigate this problem is to run the failing or crashing tests multiple times before accepting a code-removal patch.

Overall, this evidence suggests a code-removal patch always tells something interesting to the developers.

To what extent can code-removal patches, even if incorrect, give valuable information to developers to find weaknesses in test suites? (RQ3) The investigation reports that only 2 out of 48 code-removal patches (4.16%) are correct, showing that code-removal patches cannot be trusted. However, in all the cases where the patch is incorrect, 46 out of 48 cases (99.83%), the patch reveals different kinds of problems affecting the test suites that are relevant for the developers. This result is relevant to researchers, there is a need for research on exploiting code-removal patches as means to automatically improve test suites. This is also interesting for practitioners: the experiment suggests that code-removal patches should be shown to practitioners, because they can understand a particular weakness of their test suites.

3.3.4 How do developers fix the failed builds associated with a test-suite-adequate code-removal patch? (RQ4)

In this research question, the relation between the patches produced by developers and the automatically generated code-removal patches is investigated. To this end, the patches produced by the developers are first retrieved and analyzed, and then related to the code-removal patches.

Patches Produced by Developers

Table 3.7 shows the details about the relation between the failing builds for which there is a code-removal patch and the types of human patches. In particular, the first column (*Patch Type*) lists the different types of human patches associated with the builds for which jKali is able to create a patch, the second column (*Category*) shows the specific categories of every patch type, the third column (*# Builds - fail. test*), and the fourth column (*# Builds - crash. test*) indicate the number of builds with a failing or crashing test case respectively fixed according to a specific category of human fix,

TABLE 3.7: Strategies actually used by developers to fix builds patched by jKali.

Patch Type	Category	# Builds fail. test	# Builds crash. test	Tot per Category	Tot per Patch Type
Statement-Level Change	Change Condition	1	0	**1**	**2**
	Add if-else Statement	0	1	**1**	
Method-Level Change	Change Method Implementation	2	3	**5**	**7**
	Override Method	0	2	**2**	
Code Removal	Remove Variable Assignment	0	1	**1**	**5**
	Remove Variable Annotation	0	1	**1**	
	Revert	2	1	**3**	
Fix in Test	Fix Test Code	8	3	**11**	**17**
	Fix Test Data	2	4	**6**	
Not Available	No Change	5	2	**7**	**17**
	Not Found	7	3	**10**	

the fifth column (*Tot per Category*) reports the total number of builds whose patches belong to a specific category, and finally the sixth column (*Tot per Patch Type*) reports the total number of builds fixed by a specify type of human fix.

Patches fixing a single program statement is quite rare in the benchmark of this study: it happens in just 2 cases (4.17%).

A more significant number of patches span entire methods (7 cases, 14.58%). Method level changes do not follow specific patterns because they introduce or change pieces of logic in the program, they are far more complex than code-removal patches.

Interestingly, a non-trivial number of programmer patches are actually code-removal patches (5 cases, 10.42%) that remove specific program elements (e.g., assignments and annotations) or revert changes (e.g., by undoing commits or closing pull requests without merging changes). The action of reverting a change is considered like a removal of code, because the code associated with the changes is deleted with that action. This result shows that revert-based repair is relevant, while this has been little researched in academia [111].

Unexpectedly, the most frequent type of human patches target the test cases and not the application code (17 cases, 35.42%), human developer either fix the test code or the test data used by a test (e.g., a JSON file used by a test). This result reinforces the finding of RQ3 that incorrect code-removal patches can be exploited to improve test suites, including fixing wrong test cases. Moreover, it calls for more repair techniques able to generate patches for test cases and not only programs [19].

Finally, per the methodology used, the human patches are sometimes not available. In a significant number of cases, 7 (14.58%), this is because the build failure is due to flaky tests. Indeed, it is possible to notice that in 4 out of 7 cases (57.14%), the status of the build on Travis CI became `passed` after the original failure detected by Repairnator. This is a piece of evidence that techniques are needed to make sure that the build failures to be repaired are indeed not due to flakiness.

Relation between Code-removal Patches and Developers Patches

Table 3.8 relates all code-removal patches to developer patches. The first column (*Build ID*) contains the Travis CI IDs of the builds, the second column (*Build Type*) indicates if a build has a failing or crashing test case, the third column (*Code-Removal Patch Reason*) shows why a code-removal patch has been generated for a specific build, the fourth column (*Human Patch Category*) shows which is the category of patch that developers implemented to fix a bug in a particular build, and the fifth column (*Correlation Type*) indicates which type of correlation exists between the changes performed by developers patches and code-removal patches. Three different types of correlations are defined: *Same-location*, when code-removal patch and human patch change exactly the same statements of source code, 2) *Partial*, when code-removal patch and human patch have in commons at least one line of code that is changed, and 3) *Disjoint*, when code-removal patch and human patch change different points of the source code and they do not have anything in common.

Correct Patches Notably, there are two cases in which the code-removal patch is correct, build `322406277` that fails due to a failing test case, and build `384713759`, that fails due to a crashing test case. To fix the bug in build `322406277`, the developer closed the pull request refusing the change that overrides a method. The corresponding code-removal patch changes this method, forcing the execution of a specific branch, whose behavior is the same of the original method without the overriding. Thus, in this case there is a partial relation between the human patch and the code-removal patch. For build `384713759`, the developer removed an assignment to a variable, and the code-removal patch does exactly the same change. This is the only case in which the developer and the code-removal patch change exactly the same location of the source code.

Weak or Incorrect Test Cases Considering the 25 builds for which jKali is able to generate a code-removal patch because of a weak test suite, there are 4 cases in which the code-removal patches and the human patches are partially related. For build `380634197`, there is a partial relation between the changes applied by the

TABLE 3.8: Comprehensive Data of the 48 Builds with Code-removal Patches.

Build ID	Build Type	Code-Removal Patch Reason	Human Patch Category	Correlation Type
322406277	Failing test	Correct	Revert	Partial
365170225	Failing test	Weak Test Suite	Not found	None
397786068	Failing test	Weak Test Suite	Fix Test Data	None
353457987	Failing test	Weak Test Suite	Fix Test Code	None
214962527	Failing test	Weak Test Suite	Not found	None
368867994	Failing test	Weak Test Suite	Fix Test Code	None
400611810	Failing test	Weak Test Suite	Fix Test Code	None
249918159	Failing test	Weak Test Suite	Change Method Implementation	Disjoint
380634197	Failing test	Weak Test Suite	Change Method Implementation	Partial
372495757	Failing test	Weak Test Suite	Change Condition	Partial
361036711	Failing test	Weak Test Suite	Not found	None
413754623	Failing test	Weak Test Suite	Not found	None
354875355	Failing test	Rottening Test	Not found	None
403087258	Failing test	Rottening Test	Not found	None
351075282	Failing test	Rottening Test	Fix Test Data	None
378592651	Failing test	Rottening Test	Not found	None
351211949	Failing test	Buggy Test Case	Fix Test Code	None
408694507	Failing test	Buggy Test Case	Fix Test Code	None
390335750	Failing test	Buggy Test Case	Fix Test Code	None
349620528	Failing test	Buggy Test Case	Fix Test Code	None
363986485	Failing test	Buggy Test Case	Fix Test Code	None
403447416	Failing test	Flaky Test	No Change	None
415750114	Failing test	Flaky Test	Revert	Disjoint
374587117	Failing test	Flaky Test	No Change	None
402096641	Failing test	Flaky Test	No Change	None
387846982	Failing test	Flaky Test	No Change	None
415477949	Failing test	Flaky Test	No Change	None
384713759	Crashing test	Correct	Remove Assignment	Same-location
356030973	Crashing test	Weak Test Suite	Change Method Implementation	Partial
348327780	Crashing test	Weak Test Suite	Fix Test Data	None
348335601	Crashing test	Weak Test Suite	Fix Test Data	None
348337755	Crashing test	Weak Test Suite	Fix Test Data	None
356031025	Crashing test	Weak Test Suite	Not found	None
372415239	Crashing test	Weak Test Suite	Fix Test Data	None
389668297	Crashing test	Weak Test Suite	Revert	Partial
386721415	Crashing test	Weak Test Suite	Fix Test Code	None
384760371	Crashing test	Weak Test Suite	Remove Annotation	Disjoint
354919174	Crashing test	Weak Test Suite	Add if-else Statement	Disjoint
422238225	Crashing test	Weak Test Suite	Not found	None
346537408	Crashing test	Weak Test Suite	Not found	None
373018834	Crashing test	Weak Test Suite	Change Method Implementation	Disjoint
373043004	Crashing test	Weak Test Suite	Change Method Implementation	Disjoint
388971125	Crashing test	Rottening Test	Override Method	Disjoint
388971144	Crashing test	Rottening Test	Override Method	Disjoint
385681821	Crashing test	Buggy Test Case	Fix Test Code	None
363526725	Crashing test	Buggy Test Case	Fix Test Code	None
421420531	Crashing test	Flaky Test	No Change	None
415654258	Crashing test	Flaky Test	No Change	None

developer and the corresponding code-removal patch, because the code-removal patch deletes an `else` branch of the same method fixed by developers. For the build `372495757`, the human patch and the code-removal patch are partially related, because they change the same method, but in different parts. For the build `356030973`, the human patch and the code-removal patch are partially related, in this case, the code-removal patch avoids the execution of the `if` branch that is changed by developers. In the case of build `389668297`, the changes that introduce the bug have been reverted, while the corresponding code-removal patch avoids the execution of one of the paths of the new faulty method introduced by developers, indicating a partial relation between human and automated fix.

Finally, there are five cases (`249918159`, `384760371`, `354919174`, `373018834`, and `373043004`) where the code-removal patches and human patches are disjoint because they change different points of the source code. In particular, for the build `384760371`, the human patch and the code-removal patch are disjoint because they change different points of the source code, but they are semantically related because both changes influence the same value used by the program to save records in a database.

Interestingly, considering both the 25 code-removals patches that work due to weak test suites and the 7 ones that work due to buggy test cases (32 builds in total), in 16 out of 32 cases (50.0%) the human patch precisely consists in fixing the test code (11 cases) or the test data (5 cases). This confirms the deep relation between the thoroughness of the test suites and the code-removal patches that are generated.

Flaky Test When a code-removal patch works because of a flaky test, in 7 out of 8 cases (87.5%) there are no changes applied by developers to fix the failure, which is consistent.

The generation of patches that trivially alter, or do not alter at all, the semantics of the program are good indicators of failures caused by flaky tests. In fact, for the builds `403447416` and `421420531`, the code-removal patches simply force logging, without introducing any other change to the logic of the programs. For the builds `374587117` and `415654258`, the flakiness is related to timeouts. For the build `374587117`, the code-removal patch removes a piece of code not executed by the failing test case, while for the build `415654258`, the code-removal patch removes the instruction that closes a `Dispatcher` object. For the builds `402096641` and `387846982`, the code-removal patches remove an assignment instruction, while for the build `41547794`, the code-removal patch removes a method call, but apparently these actions do not influence the behavior of the program. For the remaining build `415750114`, the flakiness is related to a rare race condition that causes the failure of the test case. The corresponding code-removal patch forces the execution of a specific `if` statement, without skipping the execution of other parts of code because it is not associated with an alternative (`else`) branch.

Rottening Test The generation of code-removal patches accepted because of rottening tests can provide useful information to improve the tests, for instance by tracking the tests and the assertions that are not anymore executed after the patch. Indeed, in 2 cases (builds `354875355` and `403087258`) the code-removal patches remove the code that enables the execution of the failing test. Indeed, the test case fails only when a certain value is higher than a specified threshold. Since the code-removal patch avoids the increase of that value, the test case doesn't fail anymore. In other 2 cases, builds `388971125` and `388971144`, the developers fixed the source code overriding the method tested in the crashing test case. In these cases, the changes performed by developers and code-removal patches are disjoint.

For the build `378592651`, the code-removal patch forces the execution of a specific branch, changing a value that is used in a condition of a test case to execute certain assertions. Since the condition checks if the value is different from a given threshold before a certain time, and the code-removal patch changes that value also when it should not happen, the value satisfies the condition, and the failing assertion is not executed anymore.

Finally, in the remaining case associated with the build `351075282`, while the developer fixed the test data, the code-removal patch drops an entire block of code that influences the execution of the failing assertion in the test.

Problem of Fault Localization Overall, there are 14 builds for which the human patch changes the source code. It is observed that in 8 out of 14 cases (57.14%), the code-removal patch is at a totally different location compared to the human patch. In other terms, the fault localization technique used[14] has a poor effectiveness in those 8 cases. This is another piece of evidence that the state of the art of fault localization is under-optimal for program repair [66].

How do developers fix the failed builds associated with a test-suite-adequate code-removal patch? (RQ4) In 17 out of 48 cases (35.42%), the builds for which code-removal patches exist are caused by problems in the test suite rather than problems in the program. In particular, there are 11 out of 48 builds (22.92%) for which the developers fixed the test code, and 6 out of 48 builds (12.5%) for which the developers fixed the data used by the test cases. This is a novel observation in the literature and important for the research field: it shows that the presence of code-removal patches is a good signal about problems in tests, further confirming the results from RQ3. Also, the experiment clearly shows that fault localization often does not point to the right location to change (8 out of 14 cases, 57.14%). These results are significant for the program repair research community: this is a need to research on using the presence of code-removal patches as test adequacy criterion, and there is also a need for more research on fault localization.

[14]Ochiai in Astor/JKali, as presented in Subsection 2.1.2

3.4 Threats to validity

A threat to the validity of the results is about their generalization. Indeed, given the width of possible code-removal patches, the study considers faults revealed by either one failing or one crashing test case only, since it is a situation commonly encountered in practice. For instance, the Bears benchmark [73], which is a benchmark of 251 reproducible bugs from 72 different projects, has 71.32% of builds with a single failing (38.65% of the total) or crashing (32.67% of the total) test case. Thus, it is necessary to conduct further studies to understand if the results obtained are generalizable also to builds that have more than one failing or crashing test case.

Another threat to validity is related to the execution time chosen for the repair process, that was set to 100 minutes. To mitigate this threat, the setup of the experiment was based on previous research [25].

Another concern is about the correctness of the implementations used in the experiments. jKali, the Java implementation of Kali [95], was used to generate the code-removal patches. To mitigate this threat, both the tool and the results were made publicly available. Moreover, jKali was already used in other previous studies [76, 25, 132]

3.5 Discussion

Results show that code-removal patches are often insufficient to fix bugs, contrarily to previous studies [69, 95, 74] where the effectiveness of code-removal patches is higher. Moreover, while other approaches generically explain the presence of code-removal (or plausible) patches with the presence of a weak test suite, the presented study provides detailed evidence about issues that may affect test suites, such as rottening tests, buggy test cases, and flaky tests. The relation between code-removal patches and human patches provides additional insights about the meaning of code-removal patches. Finally, the study provides evidence that code-removal patches could be exploited to automatically improve test suites, opening new opportunities for studies in the field of program repair.

Chapter 4

Exception-Driven Fault Localization for APR

This chapter presents EXCEPT, an exception-driven fault localization technique that considers the semantics of exceptions to accurately localize faults for APR, enriching the localization with useful information about the *expressions* likely responsible for the failures and the *guessed faults*.

4.1 Fault Localization in APR

As discussed in Chapter 2, APR techniques offer a range of strategies to repair code, and all of them share the challenge of identifying the *fix locus*, that is, the program location(s) that should be modified in order to produce a fix. Indeed, it is *hard or even impossible* to repair a fault without selecting a good location for the fix [66].

The larger the program size, the more difficult it is to identify the correct location(s) in which to apply the fix: a program may consist of thousands or even millions of possible program locations, many of which executed during program failures, that can be selected for the generation of a fix. Focusing on the wrong locations may waste significant computational resources, since each location can be modified in many different ways in the attempt of obtaining a fix [71]. Even worse, choosing the wrong location multiple times can dramatically impact performance and even the feasibility of a repair attempt.

As described in Section 2.3, one of the strategies used by APR techniques to address the problem of identifying the fix locus is Spectrum Based Fault Localization (SBFL) [124]. For example, jGenProg [77], uses the ranking generated by the Ochiai SBFL technique [1] to select statements as modification points with a probability that depends on their suspiciousness, while NOPOL [131] follows the ranking generated by Ochiai to analyze each statement in the stack trace one after the other. Experimental evidence shows that SBFL techniques are often unable to rank faulty statements at top positions [66, 6, 91]

This Ph.D. thesis addresses the localization problem by primarily exploiting the *semantics of the failures* rather than the correlation between the executed statements and the crashed tests, that has been proven to produce inaccurate results. To this end, the work presented in this chapter focuses on *failures caused by exceptions*, which represent a large portion of the failures that can be observed. For instance, a study by Sawadpong et al. shows that the density of defects that are closely related to exception handling is three times higher than the overall defect density, based on six major Eclipse releases [101]. The study by Ginelli et al. considers 1,918 failing Travis CI builds discovering that 50.47% of these builds have at least one test that fails throwing an exception [31]. Finally, based on the data collected with the experiment on Repairnator [115], among the top 10 common failure reasons, 7 reasons are related to exceptions.

Exceptions carry extensive information about the occurred failures, such as the location that raised the exception, which can represent a good starting point for the fault localization, and the type of the exception, which provides useful semantic information about the possible nature of the problem. For instance, a failure caused by a Java `ArrayIndexOutOfBoundsException` suggests the location that raised the exception as a possible location for the fix, but also the statement where the array has been initialized and the statements where the variables used to access a location of the array have been assigned with a value are good locations. The type of the exception can thus be used to guide the analysis selecting suspicious locations according to the semantics of the failure.

Since locations are identified based on a guessed cause of the failure (e.g., the value of an index might be wrong), a suspicious localization can be enriched with information about the program elements that are likely responsible for the failure (e.g., a variable in an index expression) and the likely fault (e.g., wrong variable used), which can in turn be exploited to identify the change that should be operated to correct the program (e.g., replace the variable).

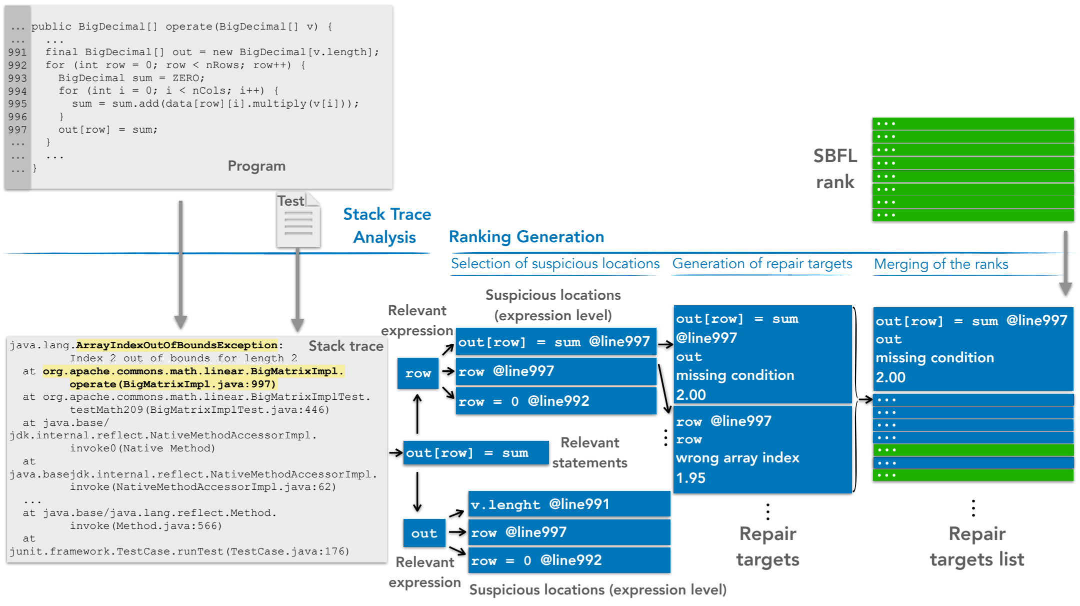
So far, some approaches have addressed the localization and repair of failures caused by `NullPointerExceptions` [23, 106], but none of them studied how to deal with multiple types of exceptions and how to enrich the localization with information that can be useful to APR techniques and developers. The localization used in ssFix [127] can exploit stack traces to improve rankings, but this is not sufficient to correctly localize several faults, as reported in the evaluation.

Although some approaches addressed the localization and repair of failures caused by exceptions, `NullPointerExceptions` in particular [23, 106], they cannot generate ranked lists of statements that can be used by APR techniques, neither they considered enriching the localization with debugging information that can be useful to developers. The localization used in ssFix [127] can exploit stack traces to improve rankings, but this is not sufficient to correctly localize several faults, as reported in the evaluation of this Ph.D. thesis.

In this context, a new fault localization technique, EXCEPT, has been developed during this Ph.D work.

4.2 Except

EXCEPT is an exception-driven fault localization technique that can be used to support APR techniques. In addition to producing an *ordered list of statements* that are reported as high-priority elements on top of ranked lists returned by SBFL techniques, EXCEPT enriches the identified items with information about the *individual expressions* that are likely faulty (e.g., a specific variable in an expression) and the *guessed faults* (e.g., wrong variable used), which can be used by both APR techniques and developers to identify the action to perform to patch the program (e.g., replacing the variable).



```
... public BigDecimal[] operate(BigDecimal[] v) {
...   ...
991   final BigDecimal[] out = new BigDecimal[v.length];
992   for (int row = 0; row < nRows; row++) {
993     BigDecimal sum = ZERO;
994     for (int i = 0; i < nCols; i++) {
995       sum = sum.add(data[row][i].multiply(v[i]));
996     }
997     out[row] = sum;
...   }
...   ...
... }
```

```
java.lang.ArrayIndexOutOfBoundsException:
          Index 2 out of bounds for length 2
  at org.apache.commons.math.linear.BigMatrixImpl.
          operate(BigMatrixImpl.java:997)
  at org.apache.commons.math.linear.BigMatrixImplTest.
          testMath209(BigMatrixImplTest.java:446)
  at java.base/
jdk.internal.reflect.NativeMethodAccessorImpl.
          invoke0(Native Method)
  at
java.basejdk.internal.reflect.NativeMethodAccessorImpl.
          invoke(NativeMethodAccessorImpl.java:62)
  ...
  at java.base/java.lang.reflect.Method.
          invoke(Method.java:566)
  at
junit.framework.TestCase.runTest(TestCase.java:176)
```

FIGURE 4.1: Except applied to the program in Listing 5.

As exemplified in Figure 4.1, EXCEPT returns a ranked list of *repair targets* starting from three inputs: a faulty program *p*, a test *t* that fails with an uncaught exception, and a list of suspicious statements *rankSBFL* identified with an SBFL technique. A repair target reports the following information:

- a program *location*, which is a likely faulty program statement,
- an *expression* in the location, which represents a specific program element likely responsible of the fault,
- *guessed faults*, which associate the expression with specific guessed faults that may affect the expression,
- a *suspiciousness value*, which is a positive number that can be used to rank repair targets from the most likely to the least likely to be relevant for fixing the fault.

For example, a repair target may refer to an array variable (expression) in a statement with an array access (program location) as likely faulty element, while suggesting that the array name is wrong (guessed fault), with a given suspiciousness value.

Or, it may identify the variable used as index of the array (expression) as the faulty element, suggesting that the value of the variable is wrong (guessed fault), with a given suspiciousness value.

EXCEPT adds the repair targets that derive from the knowledge of the exception raised by the crashing test as high priority items of the initial ranked list produced by a SBFL technique. Repair targets and SBFL locations are merged together to obtain a comprehensive ranked list that can benefit from the joint contributions of two complementary approaches. Since the information about the faulty expression and the guessed fault derives from the knowledge of the exceptions, they are available only for the high priority targets added by EXCEPT, and are not available for the elements in the initial ranked list produced by SBFL.

As depicted in Figure 4.2 EXCEPT works in two main steps:

- *Stack Trace Analysis*, which analyzes the stack trace of the exception to identify the *type of the exception* and the *relevant statements* that occur in the stack trace;
- *Ranking Generation*, which identifies the *relevant expressions* that may have caused the exception, traces them back to the *suspicious locations* that might have influenced the values of the relevant expressions, and creates a *ranked list of repair targets* from the suspicious locations and the input SBFL statements.

During the *Ranking Generation* step, a data-flow analysis is performed in order to find the locations that change the values of the relevant expressions and the locations in which the relevant expressions are used. Every relevant expression is associated with one or more suspicious locations, so as to generate different Repair Target(s) based on the type of location and the type of exception.

Figure 4.1 shows the elements that are incrementally identified by EXCEPT to finally generate the ranking, when applied to the faulty program in Listing 5, which throws an `ArrayIndexOutOfBoundsException`. Algorithm 1 and Algorithm 2 detail how EXCEPT works with pseudocode. The blue constant and the blue functions depend on the type of the exception and are described in details in Section 4.3 for the supported exceptions.

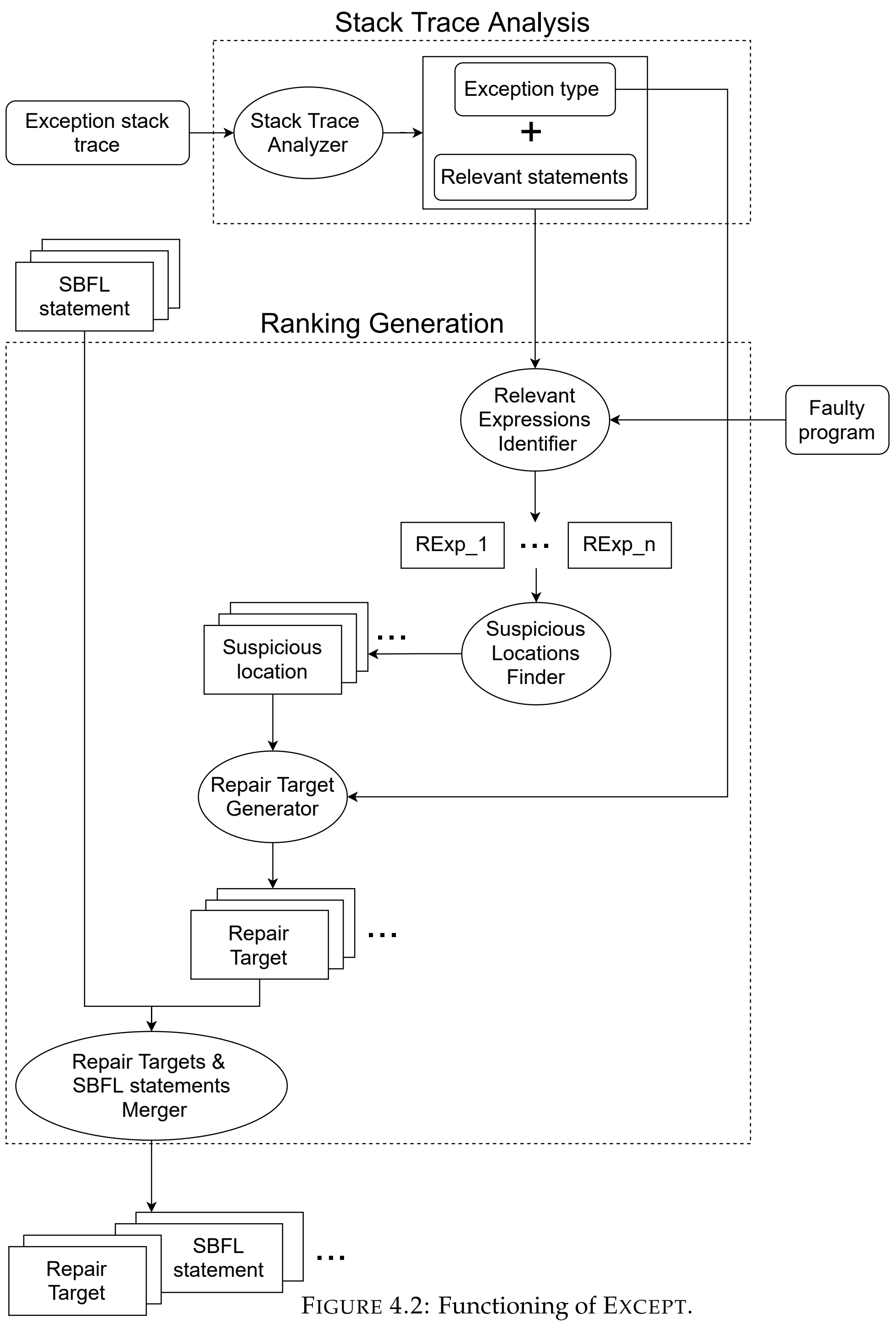


FIGURE 4.2: Functioning of EXCEPT.

4.2.1 Stack Trace Analysis

Stack trace analysis (lines 9-11 of Algorithm 1) extracts two key data from the stack trace associated with an exception: the *exception type* and the *relevant statements* for the localization of the fault. The exception type is trivial to extract. In fact, EXCEPT intercepts the output generated by the crashing test, retrieves the information about the stack trace of the exception, and extracts the line that reports the exception type.

Algorithm 1 Description of EXCEPT.

```
 1: procedure EXCEPT (p, t, rankSBFL)
 2:     Input
 3:         p           The faulty program
 4:         t           A test case that fails raising an exception
 5:         rankSBFL  The list of suspicious statements identified with a                    SBFL
    technique
 6:     Output
 7:         repairTargestList: a list of repair target or null

 8:     // Step 1: Stack Trace Analysis
 9:     stackTrace = getExceptionStackTrace(p, t);
10:     exceptionType = getExceptionType(stackTrace);
11:     relevantStatementsList = getRelevantStatementsList(stackTrace);

12:     // Step 2: Ranking Generation
13:     repairTargetsList = $\varnothing$;
14:     relevantStatementsAnalyzed = 0;

15:     // Sub-step 2.1: Selection of suspicious locations
16:     for each relevantStatement $\in$ relevantStatementsList do
17:         if relevantStatementsAnalyzed < exceptionType.maxRelevantStatementsToConsider then
18:             relevantStatementsAnalyzed = relevantStatementsAnalyzed + 1;
19:             suspiciousLocations = selectSuspiciousLocations(p, relevantStatement,
                                                                        exceptionType);
20:             // Sub-step 2.2: Generation of repair targets
21:             for each suspLoc $\in$ suspiciousLocations do
22:                 susp = computeSuspValue();
23:                 repairTarget = generateRepairTarget (suspLoc, exceptionType,            susp);
24:                 repairTargetsList.add(repairTarget);
25:             end for
26:         end if
27:     end for

28:     // Sub-step 2.3: Merging of the rankings
29:     repairTargestList.addTargetsFrom(rankSBFL);

30:     return repairTargetsList;
31: end procedure
```

EXCEPT also identifies the relevant statements that might have contributed to the exception and that should be taken under consideration to determine the possible fault locations. These statements include every program statement explicitly reported in the exception stack trace, that is, every program location that is in the context of the statement that raised the exception.

Since the fault is assumed to be in the program, EXCEPT discards from the analysis the statements that do not refer to the program under analysis, but rather refer to external libraries, JDK classes, test frameworks (e.g., JUnit [113] or Mockito [110]), and test cases. For example, considering the stack trace in Figure 4.1, EXCEPT discards the invocation to `testMath209` since the method belongs to a test case class, the invocation to `invoke0` since the method is part of the JDK library, and finally the invocation to `runTest` since the method is part of the JUnit framework.

Algorithm 2 Description of selectSuspiciousLocations.

```
procedure selectSuspiciousLocations(p, stp, et)
    Input
        p      The bugged program
        stp    A Stack Trace Poi
        et     The type of the exception

    Output
        suspiciousLocations: a set of suspicious locations

    relevantExpressions = selectRelevantExpressions(p, stp, et);

    suspiciousLocations = $\varnothing$;
    for each re $\in$ relevantExpressions do
        suspiciousLocationsForRe = findSuspiciousLocations(p, re, et);
        suspiciousLocations.add(suspiciousLocationsForRe);
    end for

    return suspiciousLocations;
end procedure
```

For each unfiltered statement, EXCEPT creates a *relevant statement* which includes the *line number* of the statement, the *Java class* to which it belongs to, the *method* that executes it, and the *file name* containing the statement. This filtering is performed by function `getRelevantStatementsList` invoked at line 11 of Algorithm 1.

The list of relevant statements is ordered according to their position in the stack trace, starting from the one closest to the statement that generates the exception under analysis.

4.2.2 Ranking Generation

The generation of the ranking (lines 13-29 in Algorithm 1) implies *1)* the analysis of the relevant statements to identify the suspicious program locations that might have caused the exception, *2)* the generation of the repair targets, and *3)* the merging of the identified repair targets with the initial SBFL rank.

Selection of Suspicious Locations (lines 16-19 in Algorithm 1 and Algorithm 2). The number of analyzed relevant statements is bound to prevent that too many repair targets are generated (line 17 in Algorithm 1). In fact, adding many targets to the initial ranking generated by SBFL may hinder the effectiveness of APR techniques, since they would have to consider too many highly suspicious program locations. On the contrary, EXCEPT aims to add a *small and focused* set of high priority repair targets that may help directing the repair algorithms on the right statements for the right reason.

In practice, the number of relevant statements to be considered might be different based on the exception type. The bound could be small (e.g., 1) for some exceptions. For instance, in the case of `ArrayIndexOfOutBoundsException`, the statement that raises the exception includes the array variable and the index value that cause the exception and the analysis can be effectively driven by their values. The bound could be higher for other exceptions, such as the `IllegalArgumentException`, since the raised exception may strongly depend on the execution context, and considering multiple points derived from the stack trace of the exception might be beneficial (e.g., also considering the calling method).

The selection of the suspicious locations from a relevant statement is described in the `selectSuspiciousLocations` function presented in Algorithm 2. The selection is driven by two key logical steps: the *selection of the relevant expressions* (line 8 in Algorithm 2) and the *identification of the suspicious locations* (line 11 in Algorithm 2).

When a relevant statement is analyzed, EXCEPT first narrows down the analysis to specific expressions included in the statement. The idea is that the analysis should focus on the relevant expressions that might be responsible for the exception, ignoring the rest of the statement. For example, if the statement that generates a `NullPointerException` is

```
p.getItem(i)
```

the analysis selects variable `p` as the expression to focus the analysis on, excluding `i` and `getItem()` from the scope of the analysis. Similarly, if a statement raises an `ArrayOutOfBoundsException`, EXCEPT would select the expression used to access the array and the expression that identifies the array as relevant expressions for the analysis, as shown in Figure 4.1. Since this step of the analysis depends on the semantics of the exception, it is described in details for each supported exception in Section 4.3.

Each expression relevant to the exception is used to identify the statements that might have caused the exceptional situation that finally resulted in the failure.

Starting from the point in which the exception occurs, the data-flow analysis searches for all the points in which the suspicious expressions are used before the statement in which the exception occurs and the points in which they are defined. All these points are then considered to generate the repair targets.

For instance, if the relevant expression is a `null` variable, the code responsible for this value would be considered suspicious at this stage. Similarly, if the relevant expressions are the array name and the array index, EXCEPT would select the code that defines the array and the code that defines the index as suspicious locations. This is done with a local data-flow analysis that depends on the relevant expression and the type of exception. The specific analysis performed for the supported exceptions is described in Section 4.3.

Generation of Repair Targets (lines 21-24 of Algorithm 1). Every suspicious location is turned into a repair target by adding the guessed faults and a suspiciousness value. The guessed faults are annotations that specify why the expression could be faulty. For example, if the exception is `ArrayIndexOutOfBoundsException` and the selected expression defines the value of the size used to initialize the array, a guessed fault may assume the initial size of the array is wrong. Developers and APR techniques can exploit this annotation to change the program accordingly. Since the annotation depends on the type of the exception, Section 4.3 describes how faults are guessed for each exception type.

The suspiciousness value assigned to the repair targets considers the fact that these targets must have higher priority compared to the top ranked elements in the input SBFL rank. Since the maximum suspiciousness in the input SBFL rank is 1, EXCEPT assigns suspicious values that start from 2 to the identified repair targets. Since repair targets are generated by following the order of occurrence of the relevant statements, which are ordered based on their distance from the statement that raises the exception, EXCEPT prioritizes the repair targets accordingly, decreasing the suspiciousness value by 0.05 every time a target is added to the rank. In practice, this guarantees that a good number of repair targets can be ranked at a higher position than the top element in the input rank.

Merging of the rankings (line 29 of Algorithm 1). The last step requires merging the identified repair targets with the input ranked list. This is performed with two simple steps. First, the locations that are present both in the SBFL rank and in the repair targets are removed, keeping only the location with the highest suspiciousness score. The additional information produced by EXCEPT (the expression and the guessed fault) are also preserved. Second, all the items are ordered according to their suspiciousness.

4.3 Supported Exceptions

To demonstrate EXCEPT, the analysis for four types of exceptions is provided: `ArrayIndexOfOutBoundsException`, `StringIndexOutOfBoundsException`, `NullPointerException`, and `IllegalArgumentException`. The focus is on some of the most popular types of exceptions based on faults contained in public benchmarks, such as Defects4J [45], Bears [73], and Repairnator [115]. Similar analyses can be added to support additional exceptions following their semantics.

EXCEPT is equipped with simple and fast data-flow-based analyses bounded in scope to identify the likely fault locations. In particular, the analysis to determine the suspicious locations is bounded to the method that includes the relevant expression. Note that the selection of the relevant expressions may select statements in multiple methods, thus the analysis is not generally limited to the method that raises the exception. Moreover, the scope of the analysis always includes the definition of class variables, which may initialize variables with wrong values. Bounding the analysis is useful to generate a limited number of repair targets and complete the analysis quickly (in the experiments every case could be processed in few seconds).

In the following, the elements that depend on the exceptions for the four supported exception types are discussed and exemplified: `maxRelevantStatementsToConsider` specifies the number of relevant statements to consider, `selectRelevantExpressions` describes how the relevant expressions are determined, `findSuspiciousLocations` describes the analysis performed to select the suspicious code locations, and finally

TABLE 4.1: Analysis of `ArrayIndexOfOutBoundsException`.

Suspicious locations	**Guessed faults**
statement with `refArray`	array variable is wrong missing conditional statement
allocation of `refArray`	wrong array initialization
definitions of the variables that determine the size of `refArray`	wrong variables values
statement with `exprIndex`	`exprIndex` is wrong
definitions of variables used in `exprIndex`	wrong variables values

`generateRepairTarget` indicates the guessed faults that are associated with the suspicious locations. The guessed fault consists of a label with known semantics (e.g., "wrong variable name") that is included in the repair target and that can be exploited by developers or APR techniques to define the repair strategy.

4.3.1 ArrayIndexOfOutBoundsException

`maxRelevantStatementsToConsider`. The analysis considers the statement that raises the exception as the only relevant statement.

`selectRelevantExpressions`. EXCEPT looks for one or more instances of the following expression in the relevant statement

    refArray[exprIndex]

to select the occurrences of `refArray` and `exprIndex` as relevant expressions that might be wrong and thus cause the exception. In fact, the access to the array might fail because the wrong array is used or the wrong array location is selected.

`findSuspiciousLocations` and `generateRepairTarget`. When `refArray` is considered, EXCEPT runs a recursive backward bounded data-flow analysis to identify the locations in which `refArray` is allocated. If the array initialization statement uses other variables, the analysis process is iterated to determine the locations that assign a value to these variables. Also the location itself where `refArray` is used is returned as a suspicious location.

When `exprIndex` is considered, EXCEPT runs a backward bounded data-flow analysis to identify the locations that define the variables that occur in `exprIndex`. Also the location itself where `exprIndex` is used is returned as a suspicious location. Table 4.1 lists the identified locations and the corresponding guessed faults.

Example. Listing 5 shows an excerpt of the Math 98 fault in Defects4J. The statement at line 9 is the relevant statement, since it generates the `ArrayIndexOutOfBoundsException`. The variables `out` and `row` are selected as *relevant expressions*, which in turn generate 6 *suspicious locations* with the corresponding guessed fault: `out` might be the wrong array variable used at line 9; `row` might be the wrong index used at line 9; there might be a missing condition (e.g., it is necessary to add an `if-statement`

```
1  public BigDecimal[] operate(BigDecimal[] v) {
2   ...
3   final BigDecimal[] out = new BigDecimal[v.length];
4    for (int row = 0; row < nRows; row++) {
5     BigDecimal sum = ZERO;
6      for (int i = 0; i < nCols; i++) {
7       sum = sum.add(data[row][i].multiply(v[i]));
8      }
9      out[row] = sum;
10    }
11   ...
12 } /*** Source: Math 98 - Defects4J ***/
```

LISTING 5: Example of `ArrayIndexOutOfBoundsException`.

that influences the access to array `out`); `row` might be assigned with the wrong value at line 4; the initialization of `out` at line 3 might be wrong; the expression `v.length` used at line 3 might be wrong. In this case, the correct patch corresponds to modifying the initialization of the array at line 3 by replacing the expression `v.length` with `nRows`, which is one of the guessed faults.

4.3.2 StringIndexOutOfBoundsException

`maxRelevantStatementsToConsider`. The analysis considers the statement that raises the exception as the only relevant statement.

`selectRelevantExpressions`. EXCEPT looks for one or more instances of the following expression in the relevant statement

`stringVar.op(...exprIndex...)`

where `stringVar` is a `String` variable, `op` is a method that can return a `StringIndexOutOfBoundsException`, such as `charAt(int index)`, and `...exprIndex...` is an Integer expression that is used to access the string at a specific position.

EXCEPT selects the occurrences of `stringVar`, in case the wrong string is used, and `exprIndex`, in case the wrong index is used, as relevant expressions.

`findSuspiciousLocations` and `generateRepairTarget`. When either `stringVar` or `exprIndex` are considered, EXCEPT runs a backward data-flow analysis to identify the locations in which the variables included in these expressions are defined. Also the location itself where these variables are used is returned as a suspicious location. Table 4.2 lists the identified locations and the corresponding guessed faults.

Example. Listing 6 shows an excerpt of the Lang 45 fault in Defects4J. The statement at line 11 is the relevant statement, since it generates the `StringIndexOutOfBoundsException`.

TABLE 4.2: Analysis of `StringIndexOutOfBoundsException`.

Suspicious locations	**Guessed faults**
statement with `stringVar`	`String` variable is wrong missing conditional statement
definition of `stringVar`	wrong value missing conditional statement
statement with `exprIndex`	`exprIndex` is wrong
definitions of `vars` *used in* `exprIndex`	wrong variables values missing conditional statement

The analysis selects the expressions `str`, `0` and `upper` as *relevant expressions*, which in turn generate 8 *suspicious locations* with the corresponding guessed fault.

`str` might be the wrong variable used at line 11, 0 might be the wrong starting index at line 11, `upper` might be the wrong end index at line 11, the entire statement at line 11 might have to be accessed only within a conditional statement, `lower` might be the wrong variable assigned as value to the variable `upper` at line 7, the entire statement at line 7 might have to be accessed only within a conditional statement, `str.length()` might be the wrong value assigned to the variable `upper` at line 4, and the entire statement at line 4 might have to be accessed only within a conditional statement.

4.3.3 NullPointerException

`maxRelevantStatementsToConsider`. The analysis considers both the statement that raises the exception and the calling method to address the case of `null` values erroneously passed as parameters.

`selectRelevantExpressions`. EXCEPT looks for one or more instances of the following expression in the first relevant statement

```
obj.op()
```

where `obj` is a non-primitive variable. EXCEPT selects the occurrences of `obj` as relevant expressions that might be wrong and thus cause the exception. Finally, EXCEPT selects the non-primitive parameters used in the method call in the second relevant statement to account for `null` values generated by the caller.

`findSuspiciousLocations` and `generateRepairTarget`. When `obj` is considered, EXCEPT runs a backward bounded data-flow analysis to identify the locations where `obj` is defined. When the caller is analyzed, if the `null` variable is defined through the method call, the calling site is also identified as a suspicious location. Also the location itself where this variable is used is returned as a suspicious location. Table 4.3 lists the identified locations and the corresponding guessed faults.

```java
public static String abbreviate(String str, int lower, int upper, ...) {
   ...
   if (upper == -1 || upper > str.length()) {
      upper = str.length();
   }
   if (upper < lower) {
      upper = lower;
   }
   ...
   if (index == -1) {
       result.append(str.substring(0, upper));
       ...
    }
    ...
} /*** Source: Lang 45 - Defects4J ***/
```

LISTING 6: Example of `StringIndexOutOfBoundsException`.

TABLE 4.3: Analysis of `NullPointerException`.

Suspicious locations	**Guessed faults**
statement with `obj`	variable is wrong missing conditional statement
definition of `obj`	wrong value missing conditional statement
calling site	wrong variables

Example. Listing 7 shows an excerpt of the Chart 4 fault included in Defects4J. The statement at line 7, since it generates the `NullPointerException`, is a relevant statement. The analysis selects the expressions `r` as *relevant expression*, which in turn generates 3 *suspicious locations* with the corresponding guessed fault: the variable `r` at line 7 might be wrong; the entire statement at line 7 might have to be accessed only within a conditional statement; and the method `getRendererForDataset()` used to assign a value to the variable `r` at line 5 might be wrong. The actual patch consists of adding the conditional statement.

4.3.4 IllegalArgumentException

`maxRelevantStatementsToConsider`. EXCEPT considers both the statement that raises the exception and the caller statement of the method that raises the exception as relevant statements for the analysis.

```
public class XYPlot {
 ...
 public Range getDataRange(ValueAxis axis) {
  ...
  XYItemRenderer r = getRendererForDataset(d);
  ...
  Collection c = r.getAnnotations();
  ...
 }
 ...
} /*** Source: Chart 4 - Defects4J ***/
```

LISTING 7: Example of `NullPointerException`.

TABLE 4.4: Analysis of `IllegalArgumentException`.

Suspicious locations	**Guessed faults**
`exprPar` *used as parameter*	wrong parameter wrong method invoked
definition of variables in `exprPar`	wrong value

`selectRelevantExpressions`. EXCEPT looks for one or more instances of the following expression in the relevant statements

`op(...exprPar...)`

which represents any method call with at least one parameter. If the invocation is found in the first relevant statement, no relevant expression is returned for the second relevant statement (i.e., the second relevant statement is skipped). Otherwise, (e.g., the first relevant statement throws the exception instead of invoking a method), the second relevant statement is also searched for the same pattern.

EXCEPT selects the occurrences of `exprPar` as relevant expressions that might be wrong, thus causing the exception.

`findSuspiciousLocations` and `generateRepairTarget`. When `exprPar` is considered, EXCEPT runs a backward bounded data-flow analysis to identify the locations where the variables used in `exprPar` are defined. Table 4.4 lists the identified locations and the corresponding guessed faults.

Example. Listing 8 shows an excerpt of the Chart 17 fault in Defect4J. The statement at line 13 raises an `IllegalArgumentException` if the value of the second parameter of method `createCopy` is less than the first one. The corresponding relevant locations are the statements at lines 13 and 4. The former statement does not contribute to the analysis since it does not include the actual invocation, while the latter does. The suspicious locations derived from the latter statement are four: the invocation of method `createCopy(int, int)`; the integer `0`; the expression `getItemCount()`

`- 1`; and the call to `getItemCount()`. The patch requires changing the expression `getItemCount() - 1` at line 4.

```
1  public class TimeSeries {
2    ...
3    public Object clone() {
4      Object clone = createCopy(0, getItemCount() - 1);
5      return clone;
6    }
7
8    public TimeSeries createCopy(int start, int end) throws ... {
9      if (start < 0) {
10        throw new IllegalArgumentException("");
11     }
12     if (end < start) {
13        throw new IllegalArgumentException("");
14     }
15     ...
16   }
17   ...
18 } /*** Source: Chart 17 - Defects4J ***/
```

LISTING 8: Example of `IllegalArgumentException`.

4.4 Empirical Evaluation

The empirical evaluation aims to answer three research questions.

RQ1 What is the fault localization effectiveness of Except?

This research question investigates the fault-localization effectiveness of Except for different types of exceptions in comparison to state of the art solutions.

RQ2 How does Except affect the capability of modifying the faulty statements of APR techniques that use SBFL?

This research question investigates how using the ranking produced by Except affects the capability of program repair techniques to modify the right statement every time the generation of a patched program is attempted.

RQ3 What is the accuracy of the guessed fault?

This research question investigates how accurately Except can guess the fault that generated an exception.

4.4.1 Empirical Setup

In the evaluation, the faults of Defects4J benchmark v1.5 [45] were used. Defects4J is a dataset of real-world bugs and patches from 5 open-source Java projects (JFreeChart, Closure Compiler, Commons Math, Joda-Time, and Commons Lang) commonly used in program repair studies [127, 74, 67].

Initially, every bug that fails with a test that raises an uncaught exception was selected. The four most frequent exception types raised by these bugs match with the classes of exceptions supported by EXCEPT: `NullPointerException` (17 faults), `IllegalArgumentException` (15 faults), `ArrayIndexOutOfBoundsException` (10 faults), `StringIndexOutOfBoundsException` (7 faults). Out of these 49 faults, the faults that require changing multiple non-contiguous code locations without having a same number of crashing tests were discarded because they could not be properly addressed by localization techniques. For instance, faults that require changing 2 locations are kept if there are 2 crashing tests that can be used to localize these changes, but the same fault is discarded if only one crashing test case is available. In total, 33 faults and 43 fault locations to be localized are considered.

Table 4.5 column *Bug ID* reports the id of the selected faults, organized by exception type. In case a fault requires changing multiple locations, the table reports multiple rows for the same Bug ID.

To answer RQ1 and RQ2, two competing approaches were considered: the Ochiai SBFL technique [1], which is the most used fault localization technique in program repair [66], and the localization strategy used in ssFix [127] (hereafter referred as *ssFix*), which exploits the entries in the stack trace as top items of the ranking. Differently from EXCEPT, ssFix does not analyze the target program to extract the relevant expressions, thus missing to select the suspicious program locations that are not in the stack trace.

To precisely measure the improvement that EXCEPT and ssFix can introduce over an existing ranking, the ranking generated by Ochiai was used as input ranking for both EXCEPT and ssFix. The test cases available with the programs were used to compute the ranking with Ochiai.

The tool implementation, which uses Spoon [90] for static program analysis, and experimental material are available at `https://gitlab.com/issta21/except`.

4.4.2 What is the fault localization effectiveness of Except? (RQ1)

In this research question, the fault localization effectiveness of Ochiai, ssFix, and EXCEPT is compared. The location modified by developers was considered as the correct fault location to be identified. If new code is added, the location next to the added code is considered as the correct one.

The metric used to compare the approaches is the *position* of the faulty statement in the rank. In the case the faulty statement has the same suspiciousness of other statements, its average position is considered, as done in other studies. That is, if

TABLE 4.5: Effectiveness results.

Bug ID	**Position**			**Probability (%)**		**Additional**
	Ochiai	ssFix	EXCEPT	Ochiai	EXCEPT	**Info**
`ArrayIndexOutOfBoundException` (AIOOBE)						
Lang 12	6.50	7.50	6.50	**6.61**	5.57	None
	6.50	7.50	6.50	**6.61**	5.57	None
Lang 61	**20.00**	21.00	21.00	**2.57**	2.31	None
Math 3	9.50	10.00	**2.00**	5.58	**17.65**	Only Target
Math 98	10.00	10.50	**2.00**	3.19	**10.43**	Both
	10.00	10.50	**2.00**	3.19	**10.43**	Both
Mockito 34	52.00	52.00	53.00	**0.30**	0.26	Both
`StringIndexOutOfBoundsException` (SIOOBE)						
Lang 6	118.50	2.00	**1.00**	0.54	**7.15**	Both
Lang 44	90.50	90.50	94.00	**0.81**	0.65	None
Lang 45	11.50	11.50	13.00	**1.22**	1.04	Only Guess
Lang 51	-	-	-	0.00	0.00	-
Lang 59	3.50	1.00	1.00	14.61	**14.85**	Both
Math 101	9.00	1.00	1.00	1.89	**8.68**	Both
`NullPointerException` (NPE)						
Chart 4	53.00	2.50	**1.00**	0.12	**0.45**	Both
Chart 14	17.50	19.50	**1.00**	0.66	**2.18**	Both
	17.50	19.50	**1.00**	0.66	**2.18**	Both
	17.50	19.50	**1.00**	0.66	**2.18**	Both
	17.50	19.50	**1.00**	0.66	**2.18**	Both
Closure 2	6.00	5.00	**1.00**	0.33	**2.63**	Both
Lang 20	23.50	23.50	**1.00**	2.43	**9.86**	Both
	8.00	11.00	**1.00**	3.47	**12.17**	Both
Lang 33	4.50	1.00	1.00	7.96	**13.74**	Both
Lang 39	27.50	1.50	**1.00**	1.92	**5.14**	Both
Lang 47	82.50	1.00	1.00	0.86	**6.78**	Both
	86.50	1.00	1.00	0.74	**6.78**	Both
Lang 57	3.50	1.00	1.00	13.46	**18.05**	Both
Math 4	4.50	**2.00**	3.00	0.82	**32.48**	Only Target
	4.50	**1.50**	8.50	0.82	**3.51**	None
Mockito 18	644.00	644.00	644.00	0.11	0.11	None
Mockito 35	1.00	1.00	1.00	0.83	0.83	None
	33.00	33.00	33.00	0.31	0.31	None
	5.00	5.00	5.00	0.69	0.69	None
Mockito 36	4.00	1.00	1.00	0.93	**5.79**	Both
Mockito 38	1.50	1.50	**1.00**	0.86	**5.87**	Both
Math 70	**1.00**	2.00	3.00	**14.04**	9.03	None
`IllegalArgumentException` (IAE)						
Chart 9	**13.50**	14.50	16.50	**1.50**	1.21	None
Chart 13	52.50	3.50	**1.00**	0.74	**1.46**	Both
Chart 17	5.00	1.50	**1.00**	2.38	**7.59**	Only Target
Chart 24	5.50	6.00	5.50	7.36	7.36	None
Closure 19	-	-	-	0.00	0.00	-
Lang 5	38.00	38.00	38.00	1.22	1.22	None
Lang 54	82.50	82.50	82.50	0.86	0.86	None
Time 27	2,636.00	2,638.00	2,636.00	**0.0037**	0.0036	None

three statements that are at the top three positions of the rank include the faulty statement, the resulting position is $\frac{1+2+3}{3} = 2$.

Table 4.5 column *Position* reports the ranking of the faulty statement for Ochiai, ssFix, and EXCEPT. When a technique performs better than the others, the cell is highlighted with light grey.

In the case of `ArrayIndexOutOfBoundsException`, EXCEPT ranked the faulty statement better than competing approaches three times. Ochiai performed better than others once (Lang 61), while ssFix was never the best approach. Interestingly, while Ochiai introduces a marginal improvement for Lang 61, EXCEPT significantly improves the ranking, moving to the second position faulty statements that were below the ninth position in the initial ranking.

In the case of `StringIndexOutOfBoundsException`, EXCEPT ranked the faulty statement better than competing approaches once, while there was never a single winning approach in the rest of the cases. In two cases, Ochiai and ssFix performed better than EXCEPT, although with a marginal improvement on the raking. On the contrary, EXCEPT significantly improved the Ochiai ranking in two cases. For this class of exceptions, ssFix and EXCEPT performed similarly. In one case about missing code (Lang 51), none of the approaches could locate the correct statement.

In the case of `NullPointerException`, EXCEPT performs significantly better than both Ochiai and ssFix both in the number of cases (10 cases) and the magnitude of the improvement. Vice versa, Ochiai and ssFix performed better than the other approaches in 1 and 2 cases, respectively, with marginal improvement in the ranking compared to EXCEPT.

Finally, in the case of `IllegalArgumentException`, EXCEPT obtained the best result in 2 cases, while Ochiai obtained the best result in 1 case, and ssFix never obtained the best result. Again, EXCEPT obtained a significative relative improvement compared to Ochiai, while the improvement of Ochiai compared to EXCEPT is marginal. Also in this case, one fault related to the addition of new code was impossible to localize.

Overall, it is noticeable how EXCEPT managed to rank well a number of faults compared to Ochiai and ssFix, with ssFix performing better than Ochiai but worse than EXCEPT. For instance, EXCEPT ranked 21 faulty statements at the first position, 26 faulty statements within the third position, and 31 faulty statements within the first 10 positions. ssfix ranked 8 faulty statements at the first position, 16 faulty statements within the third position, and 23 faulty statements within the first 10 positions. Finally, Ochiai only ranked 2 faulty statements at the first position, 3 faulty statements within the third position, and 20 faulty statements within the first 10 positions.

4.4.3 How does Except affect the capability of modifying the faulty statements of APR techniques that use SBFL? (RQ2)

To answer RQ2, it is considered how APR techniques use the rankings returned by SBFL techniques. There are two main possible models: *probabilistic* and *one-by-one*.

In the probabilistic model, APR techniques assign to each statement a probability to be selected for mutation that is proportional to its suspiciousness and the suspiciousness of the rest of the statements in the ranking:

$$prob(s) = \frac{susp(s)}{\sum\limits_{s_i \in ranking} susp(s_i)} \tag{4.1}$$

where s is a statement in the rank and *ranking* is the considered ranking. The higher the probability of selecting the faulty statement is, the more likely the APR technique will modify the right code location to fix the fault. Examples of APR techniques that use this probabilistic schema to iteratively identify the statement to be modified are jGenProg [77], jMutRepair [76], DeepRepair [122], and Cardumen [78].

In the one-by-one model, APR techniques consider the statements in the same order they occur in the ranking from the most suspicious to the least suspicious. The lower the position value of the faulty statement in the raking is, the sooner the APR technique will modify the right code location to fix the fault. Example of APR techniques that use this systematic schema to iteratively identify the statement to be modified are NOPOL [131], AE [119], and jKali [76].

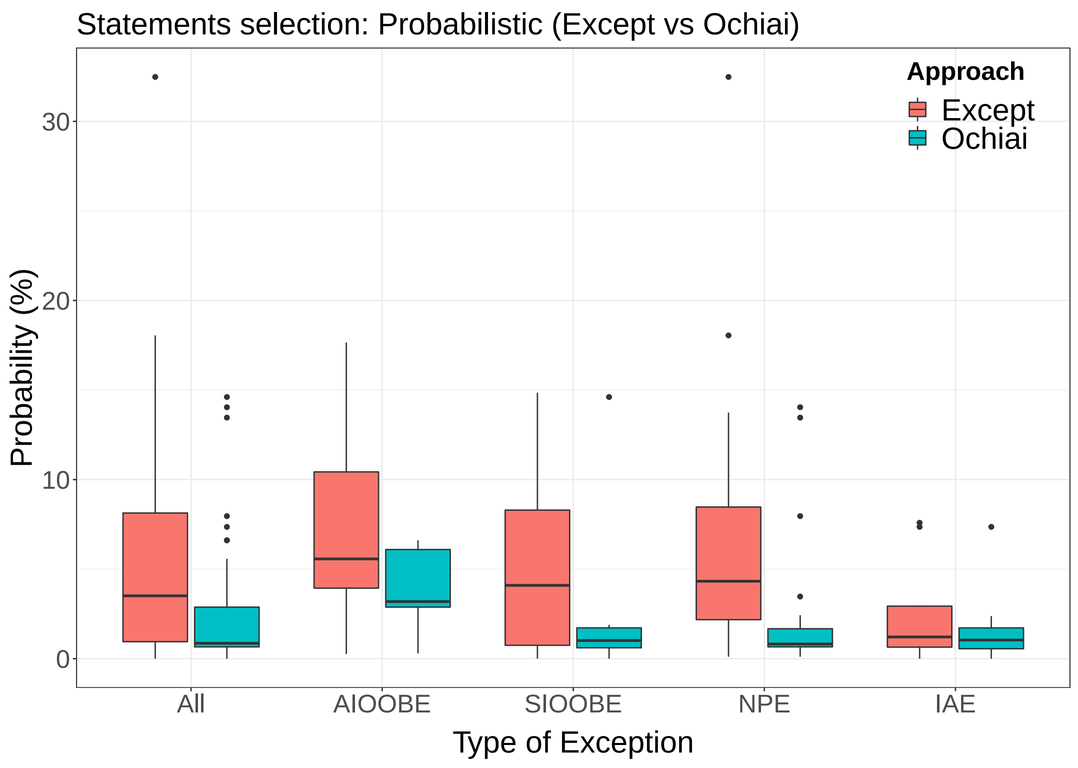


FIGURE 4.3: Comparison between Except and Ochiai according to the *probabilistic* usage of the rank.

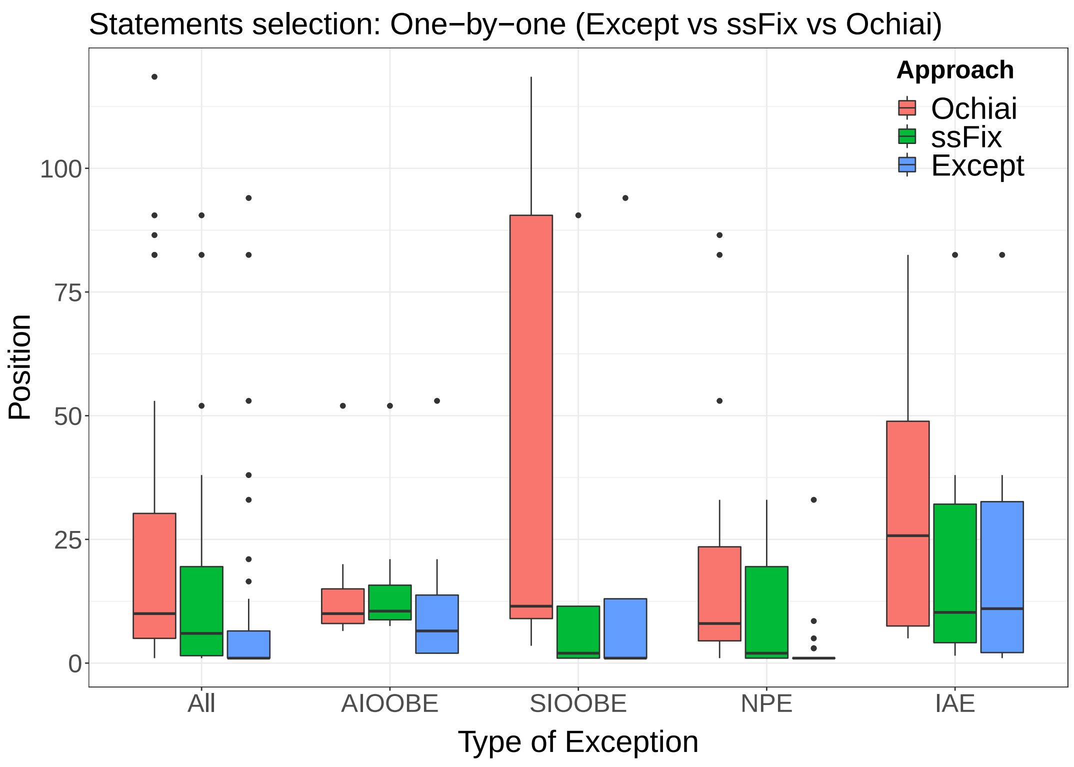


FIGURE 4.4: Comparison between Except, ssFix, and Ochiai according to the *one-by-one* usage of the rank.

To assess the impact of the rankings generated by EXCEPT, ssFix, and Ochiai in the context of APR, the metrics $prob(s)$ and $position(s)$ for all the faulty statements s considered in this study are computed. Since ssFix does not assign a probability to the statements moved at the top of the ranking, it was only possible to assess the rankings produced by ssFix with the *position* metric. The analytical values of these metrics are reported in the columns *Position* and *Probability* of Table 4.5.

Figure 4.3 visually shows the results obtained by considering the probabilistic selection schema, distinguishing per exception type and overall across all the exceptions. Probability values increase significantly with EXCEPT. In fact, the median, third quartile, and maximum obtained with EXCEPT are 3.51%, 8.14%, and 32.48% respectively, while the median, third quartile, and maximum obtained with Ochiai are 0.86%, 2.88%, and 14.61% respectively. The significant difference ($\alpha = 0.05$) between EXCEPT and Ochiai with all exceptions has been confirmed with a Wilcoxon rank sum test. The null hypothesis is that *the probability to select the correct location to fix a bug does not improve using* EXCEPT. The p-value is 0.005019, that is less than the significance level α. This means that the null hypothesis must be rejected, and thus the probability to choose the correct location can be improved using EXCEPT.

The better performance of EXCEPT is also observable at the level of the individual exception types, although with variable strength. In particular, while differences are remarkable for `ArrayIndexOutOfBoundsException`, `StringIndexOutOfBounds-Exception` and `NullPointerException`, the difference is smaller for `IllegalArgument-Exception`.

Figure 4.4 visually shows the results obtained considering the one-by-one selection schema, distinguishing per exception type and overall across all the exceptions. The relative performance of the three approaches is confirmed, with EXCEPT performing better than ssFix which performs better than Ochiai. The significant difference ($\alpha = 0.05$) between EXCEPT and both Ochiai and ssFix with all exceptions has been again confirmed with a Wilcoxon rank sum test corrected with Bonferroni correction factor. The null hypotheses is that *the correct location to fix a bug is not better ranked by* EXCEPT *compared to Ochiai and ssFix*. Comparing EXCEPT to Ochiai, the p-value is 0.00003772, while comparing EXCEPT with ssFix, the p-value is 0.01319, that are both lower than the significance level α. This means that the null hypothesis must be rejected, and EXCEPT is able to rank the correct location to fix a bug in a better way compared to Ochiai and ssFix.

At the level of the individual exceptions, it is possible to observe that EXCEPT performs better than ssFix and Ochiai for both `ArrayIndexOutOfBoundsException` and `NullPointerException`, while it performs comparably to ssFix for `StringIndexOutOfBoundsException` and `IllegalArgumentException`.

In a nutshell, the outcome of this research question suggests that faults revealed by tests that generate uncaught exceptions should be addressed by APR techniques using the ranking produced by EXCEPT, regardless of the strategy used (probabilistic or one-by-one). In fact, EXCEPT allows either to select the faulty statement with higher probability (probabilistic model) or to reach the faulty statement earlier (one-by-one model).

4.4.4 What is the accuracy of the guessed fault? (RQ3)

This research question evaluates the capability of EXCEPT to guess the fault that should be repaired. To this end, it is possible to assess the accuracy of the repair target distinguishing four main cases: both the selected expression and the guessed faults associated with the faulty statement are correct (label `Both`), the faulty expression is correctly selected but the guessed fault is wrong (label `Only Target`), the fault is correctly guessed but the wrong expression is selected (label `Only Guess`), and both the selected expression and the guessed fault are wrong (label `None`). Table 4.5 column *Additional Info* shows the results.

EXCEPT both identified the expression to be changed and the fault to be fixed in 22 out of the 37 cases with a localization (59%). In three cases the faulty expression was identified but without correctly guessing the fault. In one case, EXCEPT guesses the right fault, but associates it with the wrong statement (EXCEPT guesses a missing condition, although not in the right place). In total, EXCEPT correctly enriched the localization with additional information in 24 out of 37 cases (65%).

This result indicates that the additional information generated by EXCEPT frequently represents a meaningful suggestion about the fault.

4.4.5 Threats to validity

A threat to validity is about the limited set of exceptions supported by EXCEPT and used in the empirical evaluation. To mitigate this threat, the exceptions were selected by considering common types of problems reported in popular Java benchmarks, such as Defect4J [45], Bears [73], and Repairnator [115]. Moreover, the goal of the study is not to systematically investigate how every possible exception can be supported, but to study if fault localization can be improved by adding exception-specific support. Indeed, the study provides evidence that faulty statements can be better ranked and annotated when this kind of support is available.

Another concern is about the correctness of the implementations used in the experiments. GZoltar [14], a widely used tool, was exploited for the implementation of Ochiai. An implementation of ssFix [127] created ad-hoc for this study was used, since it was not available. Since the proposed localization schema is quite simple, both the risk of misunderstanding the technique and the risk of implementing the technique wrongly are considered small. Finally, the implementation of EXCEPT has been tested with several cases and the manual inspection of the results confirmed its correctness. To further mitigate any threats, the artefacts were made publicly available.

Finally, the main threat to external validity concerns with the generalizability of the results. To alleviate the risk of over-generalization, the study considered a number of real-world faults from different projects raising different types of exceptions. Indeed, additional experiments on other benchmarks are needed to fully address this threat.

4.5 Discussion

Results show that the ranking generated by EXCEPT for multiple classes of exceptions is more effective than the rankings generated by Ochiai and ssFix. Further, the generated ranking results in higher probability to select the faulty statements, for APR techniques that select the statements to be patched probabilistically, and in the capability to reach the faulty statement earlier, for APR techniques that select the statements to be patched one by one. EXCEPT also annotates the high priority entries of the ranking with information about the possible faults present in the code, which might be useful to developers and APR techniques.

Chapter 5

Conclusion

Software testing and debugging are difficult and expensive development activities that may take a significant portion of developers' effort, despite continuous integration and delivery practices. In this context, APR techniques may help developers in the generation of patches that can be automatically applied to programs or suggested to developers, alleviating the debugging and fixing effort.

However, APR techniques are still limited. Most of the APR techniques rely on test cases to evaluate the correctness of patches, but this is a weak validation method that admits the generation of test-suite-adequate patches that are not correct. In particular, recent empirical studies show that APR techniques, such as Kali, frequently generate code-removal patches [95, 69]. *In this context, the first contribution of this Ph.D. consists of an investigation of the factors that influence the generation of code-removal patches and the analysis of the useful information that can be extracted from code-removal patches.*

In particular, the study conducted on code-removal patches considered 1,918 failed builds with only one failing test case (950 builds) or only one crashing test case (968 builds) to determine the information that can be extracted from the code-removal patches and their relation with human patches. Thanks to the manual analysis of the patches, it has been possible to propose a comprehensive taxonomy of code-removal patches that can be exploited to better understand the current limitations of program repair techniques.

The results obtained show that code-removal patches are often insufficient to fix bugs, contrarily to previous studies [69, 95, 74] where the effectiveness of code-removal patches is higher. Moreover, while other approaches generically explain the presence of code-removal (or plausible) patches with the presence of a weak test suite, the study provides detailed evidence about issues that may affect test suites, such as rottening tests, buggy test cases, and flaky tests. The relation between code-removal patches and human patches provides additional insights about the meaning of code-removal patches. Finally, the study provides evidence that code-removal patches could be exploited to automatically improve test suites, opening new opportunities for the studies in the field of program repair.

Another key point in program repair is that if the correct location to create the patch is not identified, the generation of patches is harder or even impossible [66].

The majority of APR techniques exploit SBFL to rank the suspicious statements, although experimental evidence shows that these techniques are often unable to rank faulty statements at top positions, making the the generation of patches extremely hard. *In this context, the second contribution of this Ph.D. thesis consists of a fault localization schema that exploits the semantics of the failures to identify not only the statements, but also the expressions (e.g., variables) that are related to the failure, improving the accuracy and the amount of information returned by fault localization.*

In particular, the proposed approach focuses on the faults that are revealed by tests that fail with uncaught exceptions, which are reported as predominant in multiple benchmarks [101, 31, 115]. To effectively localize these faults, this Ph.D. thesis presents EXCEPT, a technique that exploits the semantics of the exception and the stack trace generated at the time the exception is raised, to identify a small set of highly suspicious statements that are considered with high priority by APR techniques.

Results show that the ranking generated by EXCEPT for multiple classes of exceptions is more effective than the rankings generated by competing fault localization approaches, such as Ochiai and ssFix. Further, the generated ranking results in higher probability to select the faulty statements for APR techniques that select the statements to be patched probabilistically, and in the capability to reach the faulty statement earlier for APR techniques that select the statements to be patched one by one. EXCEPT also annotates the high priority entries of the ranking with information about the possible faults present in the code, which might be useful to developers and APR techniques.

Open Challenges Based on the results obtained during this Ph.D. and presented in this thesis, it has been possible to identify new research challenges for the community. The first one is related to the fact that the presence of code-removal patches reveals different kinds of problems afflicting the test suites (not only weak test cases), and it is necessary to find a way to exploit those patches as means to automatically improve the test suites. In this way, it would be possible to potentially decrease the number of overfitting patches. The second one is that it is necessary to design a fault localization approach that exploits the semantic of failures related to the failing test cases, that do not throw an exception. Considering the promising results obtained with EXCEPT using the crashing test cases, finding a way to extract the useful information also from the failing test cases, could increase the likelihood of APR techniques using the correct locations for the generation of more correct patches.

Bibliography

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. "On the Accuracy of Spectrum-based Fault Localization". In: *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION)*. 2007.

[2] R. Abreu, P. Zoeteweij, and A. J. c. Van Gemund. "An Evaluation of Similarity Coefficients for Software Fault Localization". In: *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. 2006, pp. 39–46. DOI: `10.1109/PRDC.2006.18`.

[3] T. Ackling, B. Alexander, and I. Grunert. "Evolving Patches for Software Repair". In: *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*. GECCO '11. Dublin, Ireland: Association for Computing Machinery, 2011, 1427–1434. ISBN: 9781450305570. DOI: `10.1145/2001576.2001768`. URL: `https://doi.org/10.1145/2001576.2001768`.

[4] Amazon.com. *Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region*. Tech. rep. 2017. URL: `https://aws.amazon.com/message/41926/`.

[5] A. Arcuri. "Evolutionary repair of faulty software". In: *Applied Soft Computing* 11.4 (2011), pp. 3494–3514. ISSN: 1568-4946. DOI: `https://doi.org/10.1016/j.asoc.2011.01.023`. URL: `https://www.sciencedirect.com/science/article/pii/S1568494611000330`.

[6] F. Y. Assiri and J. M. Bieman. "Fault localization for automated program repair: effectiveness, performance, repair correctness". In: *Software Quality Journal (SQJ)* 25 (2016), pp. 171–199.

[7] F. Y. Assiri and J. M. Bieman. "MUT-APR: MUTation-Based Automated Program Repair Research Tool". In: *Advances in Information and Communication Networks*. Ed. by Kohei Arai, Supriya Kapoor, and Rahul Bhatia. Cham: Springer International Publishing, 2019, pp. 256–270. ISBN: 978-3-030-03405-4.

[8] Amazon AWS. *What is Continuous Delivery?* Tech. rep. URL: `https://aws.amazon.com/devops/continuous-delivery/`.

[9] Amazon AWS. *What is Continuous Integration?* Tech. rep. URL: `https://aws.amazon.com/devops/continuous-integration/`.

[10] B. Baudry, Z. Chen, K. Etemadi, H. Fu, D. Ginelli, S. Kommrusch, M. Martinez, M. Monperrus, J. Ron, H. Ye, and Z. Yu. *R-Hero: A Software Repair Bot based on Continual Learning*. 2020. arXiv: `2012.06824 [cs.SE]`.

[11] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen. *Reversible Debugging Software: Quantify the time and cost saved using reversible debuggers*. Tech. rep. 2013. URL: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.444.9094&rep=rep1&type=pdf`.

[12] I. Burnstein. *Practical Software Testing: A Process-Oriented Approach*. Springer Professional Computing. Springer New York, 2003. ISBN: 9780387951317.

[13] J. P. Cambronero, J. Shen, J. Cito, E. Glassman, and M. Rinard. "Characterizing Developer Use of Automatically Generated Patches". In: *2019 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2019, Memphis, Tennessee, USA, October 14-18, 2019*. Ed. by Justin Smith, Christopher Bogart, Judith Good, and Scott D. Fleming. IEEE Computer Society, 2019, pp. 181–185. DOI: `10.1109/VLHCC.2019.8818884`. URL: `https://doi.org/10.1109/VLHCC.2019.8818884`.

[14] J. Campos, A. Riboira, A. Perez, and R. Abreu. "GZoltar: an eclipse plug-in for testing and debugging". In: *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 2012, pp. 378–381. DOI: `10.1145/2351676.2351752`.

[15] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. "Pinpoint: problem determination in large, dynamic Internet services". In: *Proceedings International Conference on Dependable Systems and Networks*. 2002, pp. 595–604. DOI: `10.1109/DSN.2002.1029005`.

[16] Coralogix. *This is what your developers are doing 75% of the time, and this is the cost you pay*. Tech. rep. 2015. URL: `https://coralogix.com/log-analytics-blog/this-is-what-your-developers-are-doing-75-of-the-time-and-this-is-the-cost-you-pay/`.

[17] V. Csuvik, D. Horváth, F. Horváth, and L. Vidács. "Utilizing Source Code Embeddings to Identify Correct Patches". In: *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*. 2020, pp. 18–25. DOI: `10.1109/IBF50092.2020.9034714`.

[18] V. Dallmeier, A. Zeller, and B. Meyer. "Generating Fixes from Object Behavior Anomalies". In: *2009 IEEE/ACM International Conference on Automated Software Engineering*. 2009, pp. 550–554. DOI: `10.1109/ASE.2009.15`.

[19] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. "ReAssert: Suggesting Repairs for Broken Unit Tests". In: *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*. 2009.

[20] H. A. de Souza, M. L. Chaim, and F. Kon. *Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges*. 2017. arXiv: `1607.04347 [cs.SE]`.

[21] J. Delplanque, S. Ducasse, G. Polito, A. P. Black, and A. Etien. "Rotten Green Tests". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 500–511.

[22] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: `1810.04805 [cs.CL]`.

[23] T. Durieux, B. Cornu, L. Seinturier, and M. Monperrus. "Dynamic patch generation for null pointer exceptions using metaprogramming". In: *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2017.

[24] T. Durieux, C. Le Goues, M. Hilton, and R. Abreu. *Empirical Study of Restarted and Flaky Builds on Travis CI*. 2020. arXiv: `2003.11772 [cs.SE]`.

[25] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu. "Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, 2019, 302–313. ISBN: 9781450355728. DOI: `10.1145/3338906.3338911`. URL: `https://doi.org/10.1145/3338906.3338911`.

[26] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano. "DevOps". In: *IEEE Softw.* 33.3 (May 2016), 94–100. ISSN: 0740-7459. DOI: `10.1109/MS.2016.68`. URL: `https://doi.org/10.1109/MS.2016.68`.

[27] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. "The Daikon system for dynamic detection of likely invariants". In: *Science of computer programming* 69.1-3 (2007), pp. 35–45.

[28] Forbes.com. *Continuous Integration, Delivery, and Deployment with GitLab*. Tech. rep. 2015. URL: `https://www.forbes.com/sites/amitchowdhry/2015/05/27/new-apple-ios-bug-causes-reboots-and-messages-app-crashes/?sh=a4701123f35b`.

[29] G. Fraser and A. Arcuri. "EvoSuite: Automatic Test Suite Generation for Object-Oriented Software". In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE '11. Szeged, Hungary: Association for Computing Machinery, 2011, 416–419. ISBN: 9781450304436. DOI: `10.1145/2025113.2025179`. URL: `https://doi.org/10.1145/2025113.2025179`.

[30] L. Gazzola, D. Micucci, and L. Mariani. "Automatic Software Repair: A Survey". In: *IEEE Transactions on Software Engineering* 45.1 (2019), pp. 34–67.

[31] D. Ginelli, M. Martinez, L. Mariani, and M. Monperrus. *A Comprehensive Study of Code-removal Patches in Automated Program Repair*. 2020. arXiv: `2012.06264 [cs.SE]`.

[32] GitLab. *Continuous Integration, Delivery, and Deployment with GitLab*. Tech. rep. 2016. URL: `https://about.gitlab.com/blog/2016/08/05/continuous-integration-delivery-and-deployment-with-gitlab/`.

[33] A. Guo, X. Mao, D. Yang, and S. Wang. "An Empirical Study on the Effect of Dynamic Slicing on Automated Program Repair Efficiency". In: *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, 2018, pp. 554–558. DOI: `10.1109/ICSME.2018.00066`. URL: `https://doi.org/10.1109/ICSME.2018.00066`.

[34] Hamcrest Team. *Hamcrest*. `https://site.mockito.org`. 2021.

[35] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi. "An Empirical Investigation of Program Spectra". In: *SIGPLAN Not.* 33.7 (July 1998), 83–90. ISSN: 0362-1340. DOI: `10.1145/277633.277647`. URL: `https://doi.org/10.1145/277633.277647`.

[36] B. Hetzel. *The Complete Guide to Software Testing*. 2nd. USA: QED Information Sciences, Inc., 1988. ISBN: 0894352423.

[37] J. Hua, M. Zhang, K. Wang, and S. Khurshid. "SketchFix: A Tool for Automated Program Repair Approach Using Lazy Candidate Generation". In: ESEC/FSE 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, 888–891. ISBN: 9781450355735. DOI: `10.1145/3236024.3264600`. URL: `https://doi.org/10.1145/3236024.3264600`.

[38] J. Humble. *Continuous Delivery*. Tech. rep. URL: `https://continuousdelivery.com`.

[39] IBM. *What is DevOps?* Tech. rep. URL: `https://www.ibm.com/se-en/cloud/devops`.

[40] T. Ji, L. Chen, X. Mao, and X. Yi. "Automated Program Repair by Using Similar Code Containing Fix Ingredients". In: *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 1. 2016, pp. 197–202. DOI: `10.1109/COMPSAC.2016.69`.

[41] J. Jiang, Y. Xiong, and X. Xia. "A manual inspection of Defects4J bugs and its implications for automatic program repair". English. In: *Science China Information Sciences* 62.10 (Oct. 2019). ISSN: 1674-733X. DOI: `10.1007/s11432-018-1465-6`.

[42] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. 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*. ISSTA 2018. Amsterdam, Netherlands: Association for Computing Machinery, 2018, 298–309. ISBN: 9781450356992. DOI: `10.1145/3213846.3213871`. URL: `https://doi.org/10.1145/3213846.3213871`.

[43] J. A. Jones and M. J. Harrold. "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique". In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ASE '05. Long Beach, CA, USA: Association for Computing Machinery, 2005, 273–282. ISBN: 1581139934. DOI: `10.1145/1101908.1101949`. URL: `https://doi.org/10.1145/1101908.1101949`.

[44] R. Just, D. Jalali, and M. D. Ernst. "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. San Jose, CA, USA: Association for Computing Machinery, 2014, 437–440. ISBN: 9781450326452. DOI: `10.1145/2610384.2628055`. URL: `https://doi.org/10.1145/2610384.2628055`.

[45] R. Just, D. Jalali, and M. D. Ernst. "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. San Jose, CA, USA: ACM, 2014, pp. 437–440. ISBN: 978-1-4503-2645-2. DOI: `10.1145/2610384.2628055`. URL: `http://doi.acm.org/10.1145/2610384.2628055`.

[46] Y. Ke, K. Stolee, C. Le Goues, and Y. Brun. "Repairing Programs with Semantic Code Search". In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. 2015.

[47] D. Kelk, K. Jalbert, and J. S. Bradbury. "Automatically Repairing Concurrency Bugs with ARC". In: *Multicore Software Engineering, Performance, and Tools*. Ed. by João M. Lourenço and Eitan Farchi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 73–84. ISBN: 978-3-642-39955-8.

[48] D. Kim, J. Nam, J. Song, and S. Kim. "Automatic patch generation learned from human-written patches". In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 802–811. DOI: `10.1109/ICSE.2013.6606626`.

[49] A. J. Ko and B. A. Myers. "Debugging reinvented: asking and answering why and why not questions about program behavior". In: *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*. Ed. by Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn. ACM, 2008, pp. 301–310. DOI: `10.1145/1368088.1368130`. URL: `https://doi.org/10.1145/1368088.1368130`.

[50] R. Kou, Y. Higo, and S. Kusumoto. "A Capable Crossover Technique on Automatic Program Repair". In: *2016 7th International Workshop on Empirical Software Engineering in Practice (IWESEP)*. 2016, pp. 45–50. DOI: `10.1109/IWESEP.2016.15`.

[51] A. Koyuncu, T. F. Bissyandé, D. Kim, K. Liu, J. Klein, M. Monperrus, and Y. Le Traon. *D&C: A Divide-and-Conquer Approach to IR-based Bug Localization*. 2019. arXiv: `1902.02703 [cs.SE]`.

[52] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, and Y. 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*. ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, 2019, 314–325. ISBN: 9781450355728. DOI: `10.1145/3338906.3338935`. URL: `https://doi.org/10.1145/3338906.3338935`.

[53] H. Krasner. *The Cost of Poor Quality Software in the US: A 2018 Report*. Tech. rep. 2018. URL: `https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2018-report/The-Cost-of-Poor-Quality-Software-in-the-US-2018-Report.pdf`.

[54] S. S. S. Kruthiventi, K. Ayush, and R. V. Babu. "DeepFix: A Fully Convolutional Neural Network for Predicting Human Eye Fixations". In: *IEEE Trans. Image Process.* 26.9 (2017), pp. 4446–4456. DOI: `10.1109/TIP.2017.2710620`. URL: `https://doi.org/10.1109/TIP.2017.2710620`.

[55] S. R. Lamelas Marcote and M. Monperrus. "Automatic Repair of Infinite Loops". In: *CoRR* abs/1504.05078 (2015). arXiv: `1504.05078`. URL: `http://arxiv.org/abs/1504.05078`.

[56] J. Langr, A. Hunt, and D. Thomas. *Pragmatic Unit Testing in Java 8 with JUnit*. 1st. Pragmatic Bookshelf, 2015. ISBN: 1941222595.

[57] Q. Le and T. Mikolov. "Distributed Representations of Sentences and Documents". In: *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*. ICML'14. Beijing, China: JMLR.org, 2014, II–1188–II–1196.

[58] X. B. D. Le, D. H. Chu, D. Lo, C. Le Goues, and W. Visser. "S3: Syntax- and Semantic-Guided Repair Synthesis via Programming by Examples". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, 593–604. ISBN: 9781450351058. DOI: `10.1145/3106237.3106309`. URL: `https://doi.org/10.1145/3106237.3106309`.

[59] X. B. D. Le, D. Lo, and C. Le Goues. "History Driven Program Repair". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. 2016, pp. 213–224. DOI: `10.1109/SANER.2016.76`.

[60] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. "The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs". In: *IEEE Transactions on Software Engineering* 41.12 (2015), pp. 1236–1256.

[61] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. "GenProg: A Generic Method for Automatic Software Repair". In: *IEEE Transactions on Software Engineering* 38.1 (2012), pp. 54–72.

[62] C. Le Goues, M. Pradel, and A. Roychoudhury. "Automated Program Repair". In: *Commun. ACM* 62.12 (Nov. 2019), 56–65. ISSN: 0001-0782. DOI: `10.1145/3318162`. URL: `https://doi.org/10.1145/3318162`.

[63] Y. B. Leau, W. K. Loo, W. Y. Tham, and S. F. Tan. "Software development life cycle AGILE vs traditional approaches". In: *International Conference on Information and Network Technology*. Vol. 37. 1. 2012, pp. 162–167.

[64] C. Liu, J. Yang, L. Tan, and M. Hafiz. "R2Fix: Automatically Generating Bug Fixes from Bug Reports". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 2013, pp. 282–291. DOI: `10.1109/ICST.2013.24`.

[65] K. Liu, D. Kim, A. Koyuncu, L. Li, T. F. Bissyandé, and Y. Le Traon. "A Closer Look at Real-World Patches". In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2018, pp. 275–286. DOI: `10.1109/ICSME.2018.00037`.

[66] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon. "You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems". In: *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 2019, pp. 102–113. DOI: `10.1109/ICST.2019.00020`.

[67] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé. "TBar: Revisiting Template-Based Automated Program Repair". In: *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2019.

[68] F. Long, P. Amidon, and M. Rinard. "Automatic Inference of Code Transforms for Patch Generation". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, 727–739. ISBN: 9781450351058. DOI: `10.1145/3106237.3106253`. URL: `https://doi.org/10.1145/3106237.3106253`.

[69] F. Long and M. Rinard. "Automatic Patch Generation by Learning Correct Code". In: *SIGPLAN Not.* 51.1 (Jan. 2016), 298–312. ISSN: 0362-1340. DOI: `10.1145/2914770.2837617`. URL: `https://doi.org/10.1145/2914770.2837617`.

[70] F. Long and M. Rinard. "Staged Program Repair with Condition Synthesis". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, 166–178. ISBN: 9781450336758. DOI: `10.1145/2786805.2786811`. URL: `https://doi.org/10.1145/2786805.2786811`.

[71] F. Long and M. C. Rinard. "An analysis of the search spaces for generate and validate patch generation systems". In: *Proceedings of the International Conference on Software Engineering (ICSE)*. 2016, pp. 702–713.

[72] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. "An Empirical Analysis of Flaky Tests". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: Association for Computing Machinery, 2014, 643–653. ISBN: 9781450330565. DOI: `10.1145/2635868.2635920`. URL: `https://doi.org/10.1145/2635868.2635920`.

[73] F. Madeiral, S. Urli, M. Maia, and M. Monperrus. "Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies". In: *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '19)*. 2019. URL: `https://arxiv.org/abs/1901.06024`.

[74] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus. "Automatic Repair of Real Bugs in Java: A Large-Scale Experiment on the Defects4J Dataset". In: *Springer Empirical Software Engineering* (2016). DOI: `10.1007/s10664-016-9470-4`. URL: `https://hal.archives-ouvertes.fr/hal-01387556/document`.

[75] M. Martinez, A. Etien, S. Ducasse, and C. Fuhrman. "RTj: A Java Framework for Detecting and Refactoring Rotten Green Test Cases". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. ICSE '20. Seoul, South Korea: Association for Computing Machinery, 2020, 69–72. ISBN: 9781450371223. DOI: `10.1145/3377812.3382151`. URL: `https://doi.org/10.1145/3377812.3382151`.

[76] M. Martinez and M. Monperrus. "ASTOR: A Program Repair Library for Java". In: *Proceedings of ISSTA*. 2016. DOI: `10.1145/2931037.2948705`.

[77] M. Martinez and M. Monperrus. "Astor: Exploring the design space of generate-and-validate program repair beyond GenProg". In: *Journal of Systems and Software* 151 (2019), pp. 65 –80.

[78] M. Martinez and M. Monperrus. "Ultra-Large Repair Search Space with Automatically Mined Templates: The Cardumen Mode of Astor". In: *Search-Based Software Engineering*. Ed. by Thelma Elita Colanzi and Phil McMinn. Springer International Publishing, 2018.

[79] 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)*. 2016, pp. 691–701. DOI: `10.1145/2884781.2884807`.

[80] S. Mechtaev, J. Yi, and A. Roychoudhury. "DirectFix: Looking for Simple Program Repairs". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. 2015, pp. 448–458. DOI: `10.1109/ICSE.2015.63`.

[81] M. Monperrus. "Automatic Software Repair: A Bibliography". In: *ACM Comput. Surv.* 51.1 (Jan. 2018). ISSN: 0360-0300. DOI: `10.1145/3105906`. URL: `https://doi.org/10.1145/3105906`.

[82] M. Monperrus. *The Living Review on Automated Program Repair*. Tech. rep. hal-01956501. HAL/archives-ouvertes.fr, 2018.

[83] M. Monperrus, S. Urli, T. Durieux, M. Martinez, B. Baudry, and L. Seinturier. "Repairnator Patches Programs Automatically". In: *Ubiquity* 2019.July (July 2019). DOI: `10.1145/3349589`. URL: `https://doi.org/10.1145/3349589`.

[84] M. Motwani and Y. Brun. *Automatically Repairing Programs Using Both Tests and Bug Reports*. 2020. arXiv: `2011.08340 [cs.SE]`.

[85] G. J. Myers. *Art of Software Testing*. USA: John Wiley & Sons, Inc., 1979. ISBN: 0471043281.

[86] H. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. "SemFix: Program repair via semantic analysis". In: *Proceedings of the International Conference on Software Engineering (ICSE)*. 2013.

[87] A. Ochiai. "Zoogeographical Studies on the Soleoid Fishes Found in Japan and its Neighbouring Regions-III". In: *Nippon Suisan Gakkaishi* 22 (1957), pp. 522–525.

[88] C. Pacheco and M. D Ernst. "Randoop: feedback-directed random testing for Java". In: *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 2007, pp. 815–816.

[89] J. Pan. "Software testing". In: *Dependable Embedded Systems* 5 (1999), p. 2006.

[90] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code". In: *Software: Practice and Experience* 46 (2015), pp. 1155–1179. DOI: `10.1002/spe.2346`. URL: `https://hal.archives-ouvertes.fr/hal-01078532/document`.

[91] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller. "Evaluating and improving fault localization". In: *Proceedings of the International Conference on Software Engineering (ICSE)*. 2017.

[92] Y. Pei, Y. Wei, C. A. Furia, M. Nordio, and B. Meyer. "Code-based automated program fixing". In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. 2011, pp. 392–395. DOI: `10.1109/ASE.2011.6100080`.

[93] H. Pham. "Software Reliability". In: *Wiley Encyclopedia of Electrical and Electronics Engineering*. American Cancer Society, 1999. ISBN: 9780471346081. DOI: `https://doi.org/10.1002/047134608X.W6952`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/047134608X.W6952`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/047134608X.W6952`.

[94] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. "The Strength of Random Search on Automated Program Repair". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, 254–265. ISBN: 9781450327565. DOI: `10.1145/2568225.2568254`. URL: `https://doi.org/10.1145/2568225.2568254`.

[95] Z. Qi, F. Long, S. Achour, and M. 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*. ISSTA 2015. Baltimore, MD, USA: Association for Computing Machinery, 2015, 24–36. ISBN: 9781450336208. DOI: `10.1145/2771783.2771791`. URL: `https://doi.org/10.1145/2771783.2771791`.

[96] M. Renieres and S. P. Reiss. "Fault localization with nearest neighbor queries". In: *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* 2003, pp. 30–39. DOI: `10.1109/ASE.2003.1240292`.

[97] T. Reps, T. Ball, M. Das, and J. Larus. "The use of program profiling for software maintenance with applications to the year 2000 problem". In: *Software Engineering — ESEC/FSE'97*. Ed. by Mehdi Jazayeri and Helmut Schauer. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 432–449. ISBN: 978-3-540-69592-9.

[98] Henrique L. Ribeiro, Roberto P. A. de Araujo, Marcos L. Chaim, Higor A. de Souza, and Fabio Kon. "Jaguar: A Spectrum-Based Fault Localization Tool for Real-World Software". In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 2018, pp. 404–409. DOI: `10.1109/ICST.2018.00048`.

[99] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad. "Elixir: Effective object-oriented program repair". In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2017, pp. 648–659. DOI: `10.1109/ASE.2017.8115675`.

[100] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. "Lightweight fault-localization using multiple coverage types". In: *2009 IEEE 31st International Conference on Software Engineering*. 2009, pp. 56–66. DOI: `10.1109/ICSE.2009.5070508`.

[101] P. Sawadpong, E. B. Allen, and B. J. Williams. "Exception Handling Defects: An Empirical Study". In: *2012 IEEE 14th International Symposium on High-Assurance Systems Engineering*. 2012, pp. 90–97. DOI: `10.1109/HASE.2012.24`.

[102] Selenium Team. *Selenium*. `https://www.selenium.dev`. 2021.

[103] K. Shrestha and M. J. Rutherford. "An Empirical Evaluation of Assertions as Oracles". In: *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. 2011, pp. 110–119. DOI: `10.1109/ICST.2011.50`.

[104] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard. "Automatic Error Elimination by Horizontal Code Transfer across Multiple Applications". In: *SIGPLAN Not.* 50.6 (June 2015), 43–54. ISSN: 0362-1340. DOI: `10.1145/2813885.2737988`. URL: `https://doi.org/10.1145/2813885.2737988`.

[105] Deuslirio Silva-Junior, Plinio S. Leitao-Junior, Altino Dantas, Celso G. Camilo-Junior, and Rachel Harrison. "Data-Flow-Based Evolutionary Fault Localization". In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. SAC '20. Brno, Czech Republic: Association for Computing Machinery, 2020, 1963–1970. ISBN: 9781450368667. DOI: `10.1145/3341105.3373946`. URL: `https://doi.org/10.1145/3341105.3373946`.

[106] S. Sinha, H. Shah, C. Görg, S. Jiang, M. Kim, and M. J. Harrold. "Fault Localization and Repair for Java Runtime Exceptions". In: *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA)*. 2009.

[107] E. K. Smith, E. T. Barr, C. Le Goues, and Y. 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*. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, 532–543. ISBN: 9781450336758. DOI: `10.1145/2786805.2786825`. URL: `https://doi.org/10.1145/2786805.2786825`.

[108] Undo Software. *Increasing software development productivity with reversible debugging*. white paper. 2014. URL: `https://undo.io/media/uploads/files/Undo_ReversibleDebugging_Whitepaper.pdf`.

[109] F. Steimann, M. Frenkel, and R. Abreu. “Threats to the Validity and Value of Empirical Assessments of the Accuracy of Coverage-Based Fault Locators”. In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ISSTA 2013. Lugano, Switzerland: Association for Computing Machinery, 2013, 314–324. ISBN: 9781450321594. DOI: `10.1145/2483760.2483767`. URL: `https://doi.org/10.1145/2483760.2483767`.

[110] Szczepan Faber. *Mockito*. `https://site.mockito.org`. 2021.

[111] S. H. Tan and A. Roychoudhury. “relifix: Automated Repair of Software Regressions”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. 2015, pp. 471–482. DOI: `10.1109/ICSE.2015.65`.

[112] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury. “Anti-Patterns in Search-Based Program Repair”. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. Seattle, WA, USA: Association for Computing Machinery, 2016, 727–738. ISBN: 9781450342186. DOI: `10.1145/2950290.2950295`. URL: `https://doi.org/10.1145/2950290.2950295`.

[113] jUnit Team. *jUnit*. `https://junit.org/junit5/`. 2021.

[114] J. Troya, S. Segura, J. A. Parejo, and A. Ruiz-Cortés. “Spectrum-Based Fault Localization in Model Transformations”. In: *ACM Trans. Softw. Eng. Methodol.* 27.3 (Sept. 2018). ISSN: 1049-331X. DOI: `10.1145/3241744`. URL: `https://doi.org/10.1145/3241744`.

[115] S. Urli, Z. Yu, L. Seinturier, and M. Monperrus. “How to Design a Program Repair Bot? Insights from the Repairnator Project”. In: *Proceedings of the 40th International Conference on Software Engineering*. 2018.

[116] J. M. Voas. “A dynamic failure model for performing propagation and infection analysis on computer programs”. In: (1990).

[117] S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, and H. Jin. “Automated Patch Correctness Assessment: How Far are We?” In: *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2020, pp. 968–980.

[118] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. “Automated Fixing of Programs with Contracts”. In: *Proceedings of the 19th International Symposium on Software Testing and Analysis*. ISSTA ’10. Trento, Italy: Association for Computing Machinery, 2010, 61–72. ISBN: 9781605588230. DOI: `10.1145/1831708.1831716`. URL: `https://doi.org/10.1145/1831708.1831716`.

[119] W. Weimer, Z. P. Fry, and S. Forrest. “Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results”. In: *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Press, 2013.

[120] M. Wen, J. Chen, R. Wu, D. Hao, and S. C. Cheung. *An Empirical Analysis of the Influence of Fault Space on Search-Based Automated Program Repair*. 2017. arXiv: `1707.05172 [cs.SE]`.

[121] M. Wen, J. Chen, R. Wu, D. Hao, and S. C. Cheung. "Context-Aware Patch Generation for Better Automated Program Repair". In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, 1–11. ISBN: 9781450356381. DOI: `10.1145/3180155.3180233`. URL: `https://doi.org/10.1145/3180155.3180233`.

[122] M. White, M. Tufano, M. Martínez, M. Monperrus, and D. Poshyvanyk. "Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities". In: *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2019.

[123] J. L. Wilkerson and D. Tauritz. "Coevolutionary Automated Software Correction". In: *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*. GECCO '10. Portland, Oregon, USA: Association for Computing Machinery, 2010, 1391–1392. ISBN: 9781450300728. DOI: `10.1145/1830483.1830739`. URL: `https://doi.org/10.1145/1830483.1830739`.

[124] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. "A Survey on Software Fault Localization". In: *IEEE Transactions on Software Engineering* 42.8 (2016), pp. 707–740. DOI: `10.1109/TSE.2016.2521368`.

[125] X. Xie, T. Yueh Chen, F. C. Kuo, and B. Xu. "A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-Based Fault Localization". In: *ACM Trans. Softw. Eng. Methodol.* 22.4 (Oct. 2013). ISSN: 1049-331X. DOI: `10.1145/2522920.2522924`. URL: `https://doi.org/10.1145/2522920.2522924`.

[126] Q. Xin and S. P. Reiss. "Identifying Test-Suite-Overfitted Patches through Test Case Generation". In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2017. Santa Barbara, CA, USA: Association for Computing Machinery, 2017, 226–236. ISBN: 9781450350761. DOI: `10.1145/3092703.3092718`. URL: `https://doi.org/10.1145/3092703.3092718`.

[127] Q. Xin and S. P. Reiss. "Leveraging syntax-related code for automated program repair". In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2017, pp. 660–670.

[128] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang. "Identifying Patch Correctness in Test-Based Program Repair". In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, 789–799. ISBN: 9781450356381. DOI:

10.1145/3180155.3180182. URL: https://doi.org/10.1145/3180155.3180182.

[129] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang. "Precise Condition Synthesis for Program Repair". In: *Proceedings of the 39th International Conference on Software Engineering*. ICSE '17. Buenos Aires, Argentina: IEEE Press, 2017, 416–426. ISBN: 9781538638682. DOI: 10.1109/ICSE.2017.45. URL: https://doi.org/10.1109/ICSE.2017.45.

[130] T. Xu, L. Chen, Y. Pei, T. Zhang, M. Pan, and C. Furia. "Restore: Retrospective Fault Localization Enhancing Automated Program Repair". In: *IEEE Transactions on Software Engineering* 01 (5555), pp. 1–1. ISSN: 1939-3520. DOI: 10.1109/TSE.2020.2987862.

[131] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus. "Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs". In: *IEEE Trans. Softw. Eng.* 43.1 (Jan. 2017), 34–55. ISSN: 0098-5589. DOI: 10.1109/TSE.2016.2560811. URL: https://doi.org/10.1109/TSE.2016.2560811.

[132] He Y., M. Martinez, T. Durieux, and M. Monperrus. "A comprehensive study of automatic program repair on the QuixBugs benchmark". In: *Journal of Systems and Software* 171 (2021), p. 110825. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2020.110825. URL: http://www.sciencedirect.com/science/article/pii/S0164121220302193.

[133] D. Yang, Y. Qi, and X. Mao. "An Empirical Study on the Usage of Fault Localization in Automated Program Repair". In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2017, pp. 504–508. DOI: 10.1109/ICSME.2017.37.

[134] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan. "Better Test Cases for Better Automated Program Repair". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, 831–841. ISBN: 9781450351058. DOI: 10.1145/3106237.3106274. URL: https://doi.org/10.1145/3106237.3106274.

[135] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus. "Alleviating Patch Overfitting with Automatic Test Generation: A Study of Feasibility and Effectiveness for the Nopol Repair System". In: *Empirical Softw. Engg.* 24.1 (Feb. 2019), 33–67. ISSN: 1382-3256. DOI: 10.1007/s10664-018-9619-4. URL: https://doi.org/10.1007/s10664-018-9619-4.

[136] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus. "Test Case Generation for Program Repair: A Study of Feasibility and Effectiveness". In: *ArXiv* abs/1703.00198 (2017).

[137] A. Zeller and R. Hildebrandt. "Simplifying and isolating failure-inducing input". In: *IEEE Transactions on Software Engineering* 28.2 (2002), pp. 183–200. DOI: `10.1109/32.988498`.

[138] Z. Zhang, W.K. Chan, T.H. Tse, Y.T. Yu, and P. Hu. "Non-parametric statistical fault localization". In: *Journal of Systems and Software* 84.6 (2011), pp. 885 –905. ISSN: 0164-1212. DOI: `https://doi.org/10.1016/j.jss.2010.12.048`. URL: `http://www.sciencedirect.com/science/article/pii/S0164121211000045`.