# On Benchmarks of Bugs for Studies in Automatic Program Repair

**Fernanda Madeiral Delfim**


Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
March 2019

**Fernanda Madeiral Delfim**

# On Benchmarks of Bugs for Studies in Automatic Program Repair

Tese de doutorado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Marcelo de Almeida Maia

Uberlândia
March 2019

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema de Bibliotecas da UFU, MG, Brasil.

D349b
2019

Delfim, Fernanda Madeiral.
On benchmarks of bugs for studies in automatic program repair [recurso eletrônico] / Fernanda Madeiral Delfim. - 2019.

Orientador: Marcelo de Almeida Maia.
Tese (Doutorado) - Universidade Federal de Uberlândia, Programa de Pós-graduação em Ciência da Computação.
Modo de acesso: Internet.
Disponível em: http://doi.org/10.14393/ufu.te.2024.5079
Inclui bibliografia.
Inclui ilustrações.

1. Computação. I. Maia, Marcelo de Almeida (Orient.). II. Universidade Federal de Uberlândia. Programa de Pós-graduação em Ciência da Computação. III. Título.

CDU: 681.3

André Carlos Francisco
Bibliotecário Documentalista - CRB-6/3408

UNIVERSIDADE FEDERAL DE UBERLÂNDIA – UFU
FACULDADE DE COMPUTAÇÃO – FACOM
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO – PPGCO

The undersigned hereby certify they have read and recommend to the PPGCO for acceptance the dissertation entitled **On Benchmarks of Bugs for Studies in Automatic Program Repair** submitted by **Fernanda Madeiral Delfim** as part of the requirements for obtaining the **PhD degree in Computer Science**.

Uberlândia, 11 de Março de 2019

Supervisor: _____
Prof. Dr. Marcelo de Almeida Maia
Universidade Federal de Uberlândia

Examining Committee Members:

_____
Prof. Dr. Fabiano Cutigi Ferrari
Universidade Federal de São Carlos

_____
Prof. Dr. Flávio de Oliveira Silva
Universidade Federal de Uberlândia

_____
Prof. Dr. Leonardo Gresta Paulino Murta
Universidade Federal Fluminense

_____
Prof. Dr. Stéphane Julia
Universidade Federal de Uberlândia

SERVIÇO PÚBLICO FEDERAL
MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Ata da defesa de TESE DE DOUTORADO junto ao Programa de Pós-graduação em Ciência da Computação da Faculdade de Computação da Universidade Federal de Uberlândia.

Defesa de Tese de Doutorado: PPGCO-01/2019

Data: 11/03/2019 Hora de início: 13 :40

Discente: Fernanda Madeiral Delfin Matrícula: 11413CCP003

Título do Trabalho: On Benchmarks Of Bugs For Studies In Automatic Program Repair

Área de concentração: Ciência da Computação

Linha de pesquisa: Engenharia de Software

Reuniu-se na sala 1B132, Bloco 1B, Campus Santa Mônica da Universidade Federal de Uberlândia, a Banca Examinadora, designada pelo Colegiado do Programa de Pós-Graduação em Ciência da Computação assim composta: Professores doutores: Flávio de Oliveira Silva – FACOM/UFU, Stéphane Julia – FACOM/UFU, Fabiano Cutigi Ferrari – DC/UFSCAR, Leonardo Gresta Paulino Murta – IC/UFF e Marcelo de Almeida Maia – FACOM/UFU, orientador da candidata.

Ressalta-se que o Prof. Dr. Fábio Cutigi Ferrari participou da defesa por meio de vídeo conferência desde a cidade de São Carlos-SP e o Prof. Dr. Leonardo Gresta Paulino Murta da cidade de Niterói-RJ. Os outros membros da banca e a aluna participaram *in loco*.

Iniciando os trabalhos o presidente da mesa Prof. Dr. Marcelo de Almeida Maia apresentou a Banca Examinadora e a candidata, agradeceu a presença do público, e concedeu à Discente a palavra para a exposição do seu trabalho. A duração da apresentação da Discente e o tempo de arguição e resposta foram conforme as normas do Programa.

A seguir o senhor presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir a candidata. Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu os conceitos finais.

Em face do resultado obtido, a Banca Examinadora considerou a candidata **aprovada**.

Esta defesa de Tese de Doutorado é parte dos requisitos necessários à obtenção do título de Doutor. O competente diploma será expedido após cumprimento dos demais requisitos, conforme as normas do Programa, legislação e regulamentação interna da Universidade Federal de Uberlândia.

Nada mais havendo a tratar foram encerrados os trabalhos às 17 horas e 30 minutos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

**Participou por meio de vídeo conferência**

Prof. Dr. Fábio Cutigi Ferrari
DC/UFSCAR

**Participou por meio de vídeo conferência**

Prof. Dr. Leonardo Gresta Paulino Murta
IC/UFF

Prof. Dr. Flávio de Oliveira Silva
FACOM/UFU

Prof. Dr. Stéphane Julia
FACOM/UFU

Prof. Dr. Marcelo de Almeida Maia
FACOM/UFU(Orientador)

*To my parents.*

# Acknowledgments

My PhD journey lasted five long years. Most of it was in Brazil, but a part of it took place in France. This provided me with two different academic and cultural environments to learn from and experience. However, the path was not without challenges. I changed the topic of my thesis when I was about to start my 4th year in the PhD. Anyone who did a PhD knows what that means. Nonetheless, I learned a lot and developed many skills through tears. I also had the support of several people. Technical, financial, and, most importantly, emotional support. These people know who they are: my supervisors, collaborators, family, and friends. I will forever be grateful for their partnership and encouragement during this journey.

I also acknowledge CAPES for the financial support.

– Fernanda

*"Good research is a blameless culture, because we love taking risks in unknown territories, which is the only way to achieve true novelty."*

*– Prof. Martin Monperrus*

# Abstract

Software systems are indispensable for everyone's daily activities nowadays. If a system behaves differently from what is expected, it is said to contain a *bug*. These bugs are ubiquitous, and manually *fixing* them is well-known as an expensive task. This motivates the emerging research field *automatic program repair*, which aims to fix bugs without human intervention. Despite the effort that researchers have made in the last decade by showing that automatic repair approaches have the potential to fix bugs, there is still a lack of knowledge about the real value and limitations of the proposed repair tools. This is due to the high-level, non-advanced *evaluations* performed on the same benchmark of bugs. In this thesis, we report on contributions to the research field of automatic repair research focused on the evaluation of repair tools. First, we address the problem of the scarcity of benchmarks of bugs. We propose *a new benchmark* of real reproducible bugs, named Bears, which was built with a novel approach to mining bugs from software repositories. Second, we address the problem of the lack of knowledge about benchmarks of bugs. We present *a descriptive study* on Bears and two other benchmarks, including analyses of different aspects of their bugs. Finally, we address the problem of the extensive usage of the same benchmark of bugs to evaluate repair tools. We define *the benchmark overfitting problem* and investigate it through *an empirical study* on repair tools over Bears and the other two benchmarks. We found that most repair tools indeed overfit the extensively used benchmark. Our findings from both studies suggest that the benchmarks of bugs are complementary to each other and that the usage of multiple and diverse benchmarks of bugs is key to evaluating the generalization of the effectiveness of automatic repair tools.
**Keywords:** Software bugs. Automatic program repair. Test-suite-based repair. Repair tool evaluation. Benchmarks of bugs.

# Resumo

Sistemas de software são indispensáveis para as atividades diárias de todos hoje em dia. Se um sistema se comporta de uma maneira diferente do esperado, diz-se que ele contém um *bug*. Esses bugs são onipresentes, e *corrigi-los* manualmente é bem conhecido como uma tarefa cara. Isso motiva o campo de pesquisa emergente *reparo automático de programas*, que tem como objetivo corrigir bugs sem intervenção humana. Apesar do esforço que pesquisadores fizeram na última década, mostrando que abordagens de reparo automático têm o potencial de corrigir bugs, ainda há uma falta de conhecimento sobre o valor real e as limitações das ferramentas de reparo propostas. Isso se deve às *avaliações* de alto nível e não avançadas realizadas usando o mesmo benchmark de bugs. Nesta tese, contribuições para o campo de pesquisa de reparo automático são apresentadas, que são focadas na avaliação de ferramentas de reparo. Primeiro, o problema da falta de benchmarks de bugs é abordado. *Um novo benchmark de bugs* reais e reproduzíveis é proposto, chamado Bears, que foi construído com uma nova abordagem para mineração de bugs a partir de repositórios de software. Em seguida, o problema da falta de conhecimento sobre benchmarks de bugs é abordado. *Um estudo descritivo* sobre Bears e outros dois benchmarks de bugs é apresentado, que inclui análises sobre diferentes aspectos dos bugs. Por fim, o problema do uso extensivo do mesmo benchmark de bugs para avaliar ferramentas de reparo é abordado. *O problema de benchmark overfitting* é definido e investigado através de *um estudo empírico* sobre ferramentas de reparo usando Bears e os outros dois benchmarks. Foi descoberto que a maioria das ferramentas de reparo sofre do problema de benchmark overfitting em relação ao extensivamente usado benchmark. As descobertas a partir de ambos os estudos sugerem que os benchmarks de bugs são complementares e que o uso de múltiplos e diversos benchmarks de bugs é essencial para avaliar a generalização da eficácia das ferramentas de reparo automáticas.

**Palavras-chave:** Bugs de software. Reparo automático de programas. Reparo baseado em conjunto de testes. Avaliação de ferramenta de reparo. Benchmarks de bugs.

# List of Figures

# List of Tables

# List of Algorithms

# Listings

# Contents

Chapter 1

# Introduction

*Software bugs* started to exist when the first software system was created. The existence of bugs decreases the quality of software systems from the point of view of their users: users do not care about how a system is implemented or if it is easy to maintain; however, they care if a system does not behave as expected. Ideally, all bugs in a system should be *fixed* before releasing the system. However, software systems constantly evolve, and code change might result in more bugs. This is especially notable because new software system releases usually include, in addition to new functionalities and code improvements, *bug fixes*. Therefore, in each development iteration, bugs are fixed, and new ones appear.

*Bug fixing* is an important activity to improve the quality of software systems. This activity is also well-known to be difficult and time-consuming. This is due to the several tasks that usually need to be performed until a fix is reached, such as the detection of the unexpected behavior (i.e., verification of the existence of a bug), the localization of the root cause of the bug (i.e., fault localization), and the modification of the program source code to make it work as expected (i.e., bug repair) (MONPERRUS, 2014).

The software engineering research area includes several fields dedicated to supporting developers in activities related to bugs. This support ranges from techniques for writing tests for bug detection to automated support for debugging systems to find the root cause of a bug. Despite great progress in these bug-related fields over the last decades, the construction of a fully automated solution for fixing bugs is still challenging. In particular, modifying the source code to fix a bug is still written by a developer.

More recently, an ambitious research field has emerged that aims to automate the bug fixing activity, named *automatic program repair*. Automatic program repair consists of automatically finding solutions to software bugs, without human intervention (MONPERRUS, 2018a). Unlike recommendation systems for fixing bugs, which typically recommend understandable, but partial solutions for a human filling the gap, automatic repair systems have no understandability constraints but must always provide solutions 100% executable (MONPERRUS, 2014).

The biggest family of automatic program repair approaches is the *test-suite-based* one, which is the focus of this thesis. A test-suite-based repair approach uses the test suite of the program under repair as the *specification* of the behavior expected from the program. This specification must contain two oracles: the bug oracle and the regression oracle. The former is about the specification of the current unexpected behavior caused due to a bug, which is composed of at least one failing test case. The latter is about the specification of the other behaviors expected from the program, which is composed of passing test cases. Then, the problem statement for test-suite-based program repair is *given a program and its test suite with at least one failing test case, create a patch that makes the whole test suite pass* (MONPERRUS, 2014; MONPERRUS, 2018a).

## 1.1 Problem Statement

The literature on automatic program repair is broad on repair approaches and tools, to the extent that the field has grown fast. In this thesis, instead of proposing a new repair approach and tool, we explore a different path by going back to the foundations of the field: the *evaluation* of approaches and tools. Our motivation is based on the fact that the potential value of existing repair tools, and also of new repair tools to be further proposed, can only be measured through well-conducted empirical evaluations. These evaluations are conducted by running repair tools on *known bugs*. Each known bug consists of a buggy program version and a mechanism to expose the bug, such as a failing test case. Some evaluations are ad-hoc, but the most valuable ones from a scientific point of view are based on *benchmarks of bugs* built using systematic approaches. However, there are several open problems related to benchmarks of bugs and, by extension, to the evaluation of repair tools.

**Problem #1:** *The scarcity of benchmarks of bugs.* There is a limited number of benchmarks of bugs for automatic program repair in the Java programming language. Defects4J (JUST et al., 2014) and Bugs.jar (SAHA et al., 2018) are the most suitable ones, which contain bugs mined in the wild from real Java programs. There should not be a problem if these benchmarks are not biased. However, no benchmark is perfect (LE GOUES et al., 2015). Benchmarks should be representative of bugs and projects from which they come. The extent to which the existing benchmarks are representative is unknown because even the distribution of the real-world bugs is unknown. A way to alleviate this problem is to build new complementary benchmarks with well-designed approaches that can also be further evolved by the research community.

**Problem #2:** *The lack of knowledge about benchmarks of bugs.* Building benchmarks of bugs is a challenging task. Despite the effort made by authors of bug benchmarks, such benchmarks usually do not include detailed information about the bugs. The lack

of information is an obstacle to the progress of research in automatic program repair. A way to alleviate this problem is to conduct studies on the bugs in benchmarks, including, for instance, their categorization according to the way in which they were fixed by developers, such as repair actions performed in the source code. This type of study results in knowledge that can be useful for helping researchers 1) measure the representativeness of the benchmarks and find their flaws, 2) learn about real bugs and, by extension, improve their repair approaches and tools, and 3) perform bug sampling and advanced analysis on evaluations of repair tools.

**Problem #3:** *The extensive usage of the same benchmark of bugs to evaluate automatic program repair tools.* Through a review of the literature on evaluations of repair tools for Java, we found that ∼80% of the 24 test-suite-based repair tools are proposed based on an evaluation on the Defects4J benchmark only. This is a major threat to the validity of these evaluations because the extent to which Defects4J represents all types of bugs and their distribution in real systems is unknown. We hypothesize that repair tools might overfit Defects4J and not work as well on other real-world bugs, making it difficult to draw conclusions about their potential. To alleviate this problem, repair tools should be evaluated on different benchmarks of bugs.

## 1.2 Contributions

This thesis aims to address the three problems stated above. Figure 1 shows an overview of the contributions (highlighted in bold) in the context of existing works in the field. The three main contributions are explained as follows.

**Contribution #1 [A benchmark]:** To address problem #1, we created Bears, a project to collect and store bugs in an extensible bug benchmark for automatic repair studies in Java. The collection of bugs relies on commit building state from Continuous Integration (CI) to find potential pairs of buggy and patched program versions from open-source projects hosted on GitHub. Each pair of program versions passes through a pipeline in which an attempt is made to reproduce a bug and its patch. The core step of the reproduction pipeline is the execution of the test suite of the program on both versions of the program. If a test failure is found in the buggy program version candidate and no test failure is found in the patched program version candidate, a bug and its patch were successfully reproduced. The uniqueness of Bears is the usage of CI builds to identify buggy and patched program version candidates, which has been widely adopted in open-source projects in the last few years. This approach allows us to collect bugs from a diversity of projects beyond mature ones that use bug tracking systems. Moreover, Bears was designed to be publicly available and easily extended by the research community through the automatic creation of branches with bugs in a given

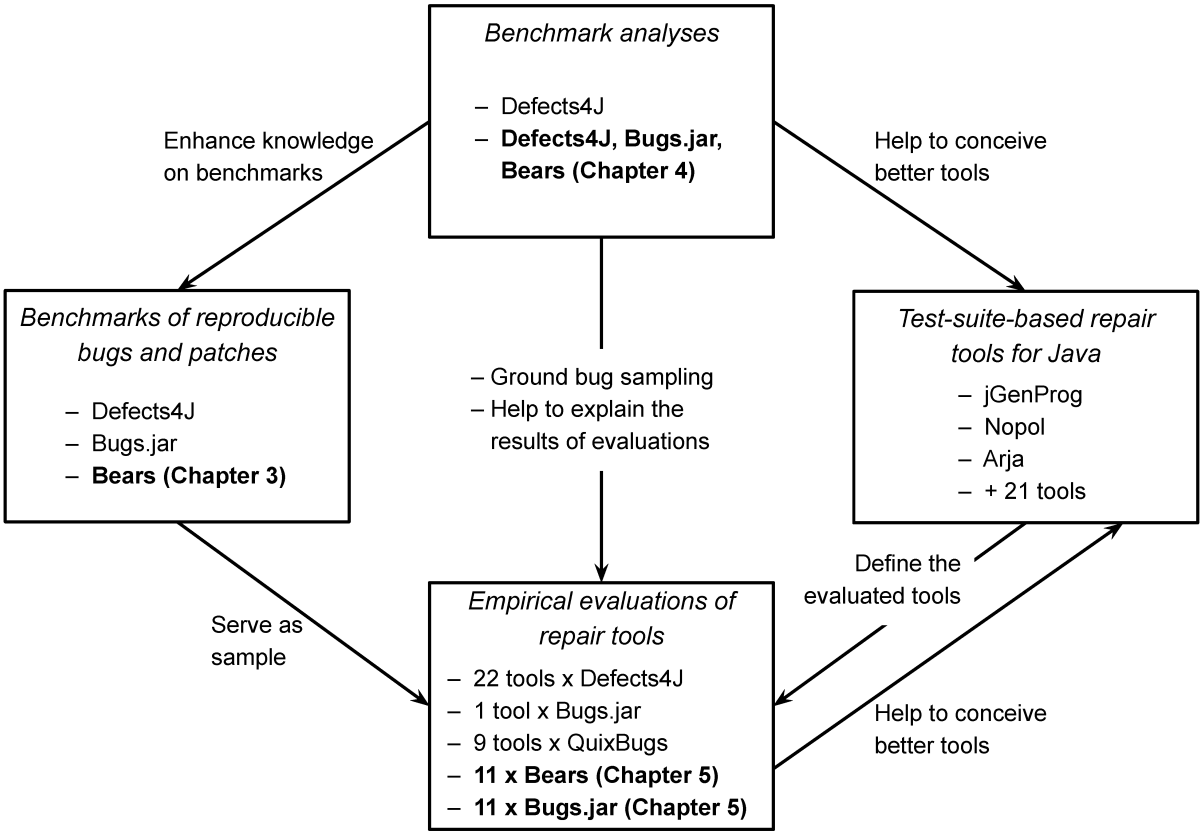


Figure 1 – Overview of the contributions of this thesis in the context of the existing works. Our contributions are highlighted in bold.

GitHub repository, which can be used for pull requests in the Bears repository. In this thesis, we deliver version 1.0 of Bears, which contains 251 reproducible bugs collected from 72 projects that use the Travis CI and Maven build environment.

**Contribution #2 [A study on benchmarks]:** To address problem #2, we conducted a descriptive study on Bears and the other two benchmarks of real bugs, Defects4J and Bugs.jar. This study includes the collection and analysis of information on bugs according to three aspects: the projects from which the bugs were mined, their exposure by failed test cases, and the patches written by developers that fixed them. To do so, we performed manual and automated analyses on the benchmarks of bugs.

**Contribution #3 [A study on repair tools]:** To address problem #3, we conducted a large-scale empirical study that includes the execution of 11 test-suite-based repair tools on three different benchmarks of bugs: Defects4J, Bugs.jar, and Bears. The primary goal of this study is to investigate whether existing repair tools behave in a similar way on different benchmarks of bugs. If a repair tool performs significantly better on one benchmark than on other ones, we conclude that the repair tool *overfits the benchmark*.

## 1.3 Thesis Organization

The remainder of this thesis is structured in the following chapters.

**Chapter 2: Background & State of the Art.** This thesis lies in the field of automatic program repair. In Chapter 2, we provide key concepts as well as a literature review to ground the problem statement of this thesis.

**Chapter 3: BEARS: An Extensible Java Bug Benchmark for Automatic Program Repair Studies.** In Chapter 3, we present BEARS, aiming to address the scarcity of benchmarks of bugs (problem #1).

**Chapter 4: A Descriptive Study on Bug Benchmarks.** In Chapter 4, we present a descriptive study on benchmarks of bugs, aiming to address the lack of knowledge about benchmarks of bugs (problem #2).

**Chapter 5: An Empirical Study on Repair Tools versus Bug Benchmarks.** In Chapter 5, we present an empirical study on repair tools on different benchmarks of bugs, aiming to address the extensive usage of Defects4J to evaluate automatic program repair tools (problem #3).

**Chapter 6: Conclusion.** We conclude this thesis by first revisiting the scientific contributions. Then, we summarize the artifacts related to this thesis and the bibliographical contributions. Finally, we elaborate on future works and present our last words.

# Chapter 2

# Background & State of the Art

A software system is meant to meet a *specification* of the behavior expected by the users of the system. When the system behaves differently from what is expected, it is said to contain a *bug*. There are several terms in Software Engineering that are related to the term bug: fault, error, and failure. A *failure* is an unexpected behavior observed by the user—according to Avizienis et al. (2004), a failure occurs either because the system does not comply with the specification or because the specification does not adequately describe the expected behavior of the system. Before a failure, there was an *error* in the system, which is a propagating incorrect state that has not yet been noticed, and the root cause of the error is a *fault*, which is an incorrect code (MONPERRUS, 2018a). In this thesis, we use the term *bug* to refer to an incorrect system source code that causes a failure[1].

Software bugs are ubiquitous (LE GOUES et al., 2012) and their existence decreases the quality of software systems from the point of view of users. However, manually fixing them is well known to be a difficult and time-consuming task. This motivates the design and development of techniques to automatically find solutions to software bugs, which is the goal of the research field known as *automatic program repair*. This chapter is dedicated to providing a general background on automatic program repair, as well as a review of the literature to support the problem statement presented in Section 1.1. First, we present general concepts on automatic program repair, followed by a review of repair tools that are based on the type of technique of our interest, named *test-suite-based automatic program repair* (Section 2.1). Second, we review existing assets to evaluate repair tools, which are the benchmark of bugs (Section 2.2). Finally, we put both previous topics together by presenting the existing evaluations of repair tools on benchmarks of bugs (Section 2.3).

[1] According to Monperrus (2018a), "*there is absolutely no emerging separation between automatic repair of failures, automatic repair of errors, and automatic repair of faults*".

## 2.1 Automatic Program Repair

Automatic program repair is the research field that aims to automatically find solutions to software bugs, without human intervention (MONPERRUS, 2018a). There are two main ways for automatically repairing programs: *behavior repair* and *state repair*[2]. These two groups are the first tree level in the taxonomy of automatic program repair techniques illustrated in Figure 2.

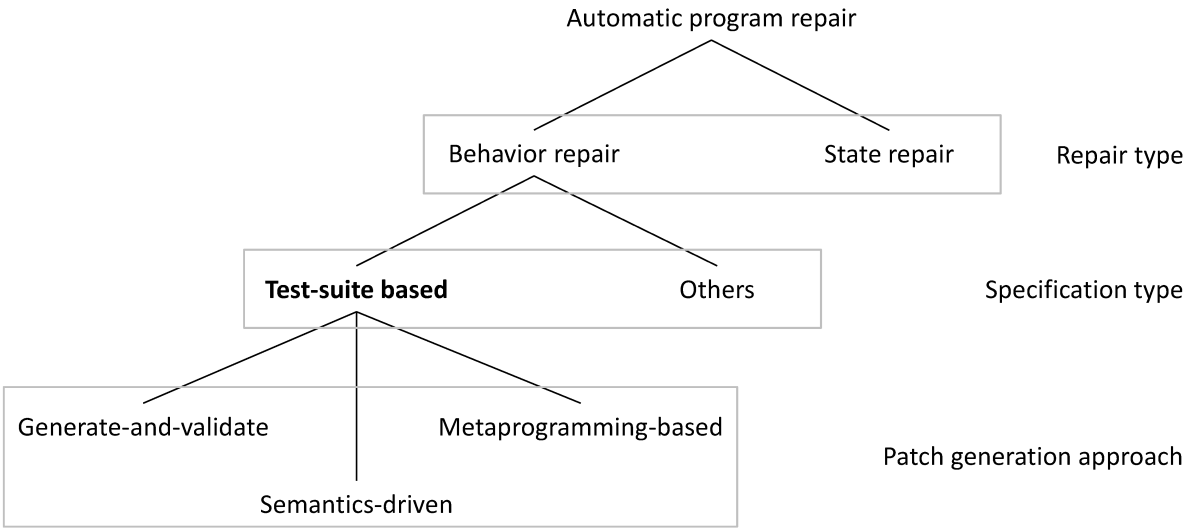


Figure 2 – Taxonomy of automatic program repair techniques, adapted from Liu et al. (2018).

In order to explain a behavior repair technique and a state repair technique, consider the toy, illustrative example in Figure 3: the left side contains a Java source code snippet from a hypothetical application, and the right side contains a test case for this application. The test case exercises the method `methodX(Object object)` from `ClassA`. One can observe that the reference `object` is set to `null` in the test case, and it is not changed until the execution of the statement `object.methodY();` in the `ClassA`. During the execution of the test, an exception is thrown in the normal operation of the Java Virtual Machine when such a statement is reached, crashing the program. This happens because there is no handler for `object` before accessing it.

One way to avoid such an exception is to change the *code* by, for example, adding an `if` condition to check if `object` is null, and if so, to skip the statement that accesses `object`: repair by changing the code is *behavioral repair*. Another way to avoid the crashing of the program is to change the *state* of the program by, for example, replacing the null value of `object` for an existing non-null value at runtime: repair by changing the state of the program is *state repair*.

[2] We adopted the terms *behavior repair* and *state repair* from Monperrus (2018a), which are referred to as *software repairing* and *software healing*, respectively, by Gazzola et al. (2019).

```
public class ClassA {
    [...]
    public void methodX(Object object) {
        object.methodY();
    }
    [...]
}
```

```
@Test
public test_object_null() {
    ClassA classA = new ClassA();
    Object object = null;
    classA.methodX(object);
}
```

Figure 3 – An example of null pointer exception.

### 2.1.1 Behavioral Repair Process

The type of repair that we are interested in is *behavioral repair*, i.e., repair by changing the source code of the system under repair. Gazzola et al. (2019) presented a high-level overview of the behavioral repair process, which we use in this section with some terminology adaptations. Figure 4 presents this overview, which starts with a buggy program. The *failure detection* step is responsible for classifying executions as either failures or failure-free executions. These executions are then processed to identify and fix the bugs that originated them.

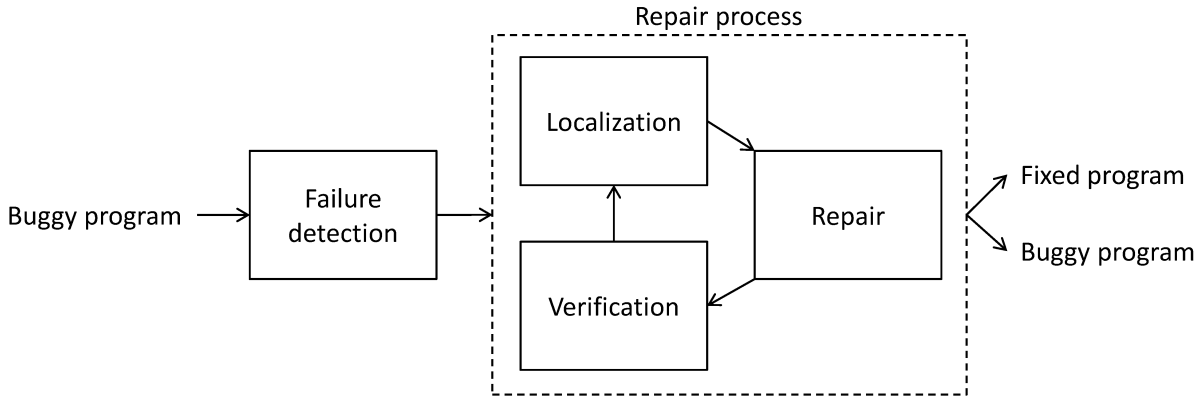


Figure 4 – Automatic behavioral program repair process, adapted from Gazzola et al. (2019).

The behavioral repair process itself is then composed of three steps. The *localization* step identifies the locations where a fix could be applied: these locations are referred to as *suspicious statements*. Then, the *repair* step generates fixes that modify the program in the code locations returned by the localization step, named *patch*. Finally, the *verification* step checks if the created patch has actually repaired the program. The *repair* and *verification* steps might be iterated several times for multiple locations until the bug has been fixed, no further patches can be generated, or the time allocated to the repair process has been exhausted. The repair process might produce two outcomes: (i) *fixed program*, which means that the program has been fixed and the fixed program will not produce further failures (successful repair), or (ii) *buggy program*, which means that the program is still buggy (unsuccessful repair).

### 2.1.2 Test-suite-based Automatic Program Repair

The repair process relies on a *specification* of the program, which is used to check the correctness of a repair attempt. This work focuses on repair combined with a *test suite* as a specification (see the second tree level in Figure 2). This combination is called *test-suite-based repair* (MONPERRUS, 2014), and it turns the problem statement of behavioral repair in "*given a program and its test suite with at least one failing test case, create a patch that makes the whole test suite pass*" (MONPERRUS, 2018a). Typically, the test suite includes at least one test case that triggers the bug (i.e., a failing test case) and tests of the expected behavior (i.e., passing ones) (LE GOUES et al., 2012).

There are several test-suite-based automatic program repair tools in the literature. In this section, we present a non-exhaustive review of repair tools—for a complete view, we refer to the two existing surveys on automatic program repair (MONPERRUS, 2018a; GAZZOLA et al., 2019). Since the focus of this thesis is on benchmarks of bugs for repair tools for Java programs, we selected all test-suite-based repair tools for Java from the living review on automatic program repair maintained by Monperrus (2018b). Table 1 shows the selected tools sorted by publication year as an overview of the progress of the test-suite-based tools for Java.

The repair tools are classified into three categories (see the third tree level in Figure 2): *generate-and-validate*, *semantics-driven*, and *metaprogramming-based*. *Generate-and-validate* approaches generate populations of possible patch candidates by heuristically modifying program Abstract Syntax Trees and then validate these patch candidates with the test suite of the program under repair. *Semantics-driven* approaches leverage symbolic execution and test suites to extract semantic constraints for the behavior under repair, and then use program synthesis to generate patches that satisfy the extracted constraints. The third category of approaches, *metaprogramming-based*, includes approaches that first create a metaprogram of the program under repair and then explore it at runtime, which in the end uses the runtime information to generate patches.

We selected 11 repair tools from Table 1 to be described in this chapter. These repair tools are the ones we used in our study presented in Chapter 5. The selection criteria are described in Section 5.1.2.

**jGenProg (MARTINEZ; MONPERRUS, 2016) and GenProg-A (YUAN; BANZHAF, 2018).** jGenProg and GenProg-A are implementations of GenProg (WEIMER et al., 2009), which is a seminal repair tool for C programs, for Java programs. GenProg is a redundancy-based repair approach (MARTINEZ et al., 2014) that generates patches using existing code (i.e., the *ingredient*) from the system under repair, i.e., it does not synthesize new code. GenProg works at the statement level, and the repair operations are the insertion, removal, and replacement of statements. The replacement operator replaces one statement with another of the same type, e.g., an assignment is only replaced by another assignment.

Table 1 – Test-suite-based program repair tools for Java sorted by publication year.

| Repair tool (24) | Patch generation approach |
|---|---|
| PAR (KIM et al., 2013) | Generate-and-validate |
| jGenProg (MARTINEZ; MONPERRUS, 2016) | Generate-and-validate |
| jKali (MARTINEZ; MONPERRUS, 2016) | Generate-and-validate |
| jMutRepair (MARTINEZ; MONPERRUS, 2016) | Generate-and-validate |
| HDRepair (LE et al., 2016) | Generate-and-validate |
| xPAR (LE et al., 2016) | Generate-and-validate |
| Nopol (XUAN et al., 2016) | Semantics-driven |
| DynaMoth (DURIEUX; MONPERRUS, 2016a) | Semantics-driven |
| NPEFix (DURIEUX et al., 2017) | Metaprogramming-based |
| ACS (XIONG et al., 2017) | Generate-and-validate |
| ELIXIR (SAHA et al., 2017) | Generate-and-validate |
| JAID (CHEN et al., 2017) | Generate-and-validate |
| ssFix (XIN; REISS, 2017b) | Generate-and-validate |
| CAPGEN (WEN et al., 2018) | Generate-and-validate |
| Cardumen (MARTINEZ; MONPERRUS, 2018) | Generate-and-validate |
| LSRepair (LIU et al., 2018b) | Generate-and-validate |
| SimFix (JIANG et al., 2018) | Generate-and-validate |
| SKETCHFIX (HUA et al., 2018) | Generate-and-validate |
| SOFix (LIU; ZHONG, 2018) | Generate-and-validate |
| ARJA (YUAN; BANZHAF, 2018) | Generate-and-validate |
| GenProg-A (YUAN; BANZHAF, 2018) | Generate-and-validate |
| Kali-A (YUAN; BANZHAF, 2018) | Generate-and-validate |
| RSRepair-A (YUAN; BANZHAF, 2018) | Generate-and-validate |
| DeepRepair (WHITE et al., 2019) | Generate-and-validate |

**jKali (MARTINEZ; MONPERRUS, 2016) and Kali-A (YUAN; BANZHAF, 2018).** jKali and Kali-A are implementations of Kali (QI et al., 2015) for Java programs. Kali performs repair by removing or skipping code. The operators implemented in Kali are removal of statements, modification of `if` conditions to *true* and *false*, and insertion of `return` statements.

**jMutRepair (MARTINEZ; MONPERRUS, 2016).** jMutRepair is an implementation of the mutation-based repair approach presented by Debroy and Wong (2010) for Java programs. It considers three kinds of mutation operators: relational (e.g., `==`), logical (e.g., `&&`), and unary (i.e., addition or removal of the negation operator `!`). jMutRepair performs mutations on these operators in suspicious `if` condition statements.

**Cardumen (MARTINEZ; MONPERRUS, 2018).** Cardumen is a test-suite-based repair tool that synthesizes new expressions to replace suspicious expressions. Cardumen mines templates (i.e., piece of code at the expression level, where the variables are replaced by placeholders) from the code under repair. Then, Cardumen selects a template that is compatible with a suspicious expression *se*, and creates a new expression from it by replacing all its placeholders with variables frequently used in the context of *se*.

**ARJA (YUAN; BANZHAF, 2018).** ARJA is a genetic programming tool that optimizes the exploration of the search space by combining three different approaches: a patch representation for decoupling properly the search subspaces of likely-buggy locations, operation types and ingredient statements; a multi-objective search optimization for minimizing the weighted failure rate and for searching simpler patches; and a method/variable scope matching for filtering the replacement/inserted code to improve compilation rate.

**RSRepair-A (YUAN; BANZHAF, 2018).** RSRepair-A is an implementation of RSRepair (QI et al., 2014) for Java programs. RSRepair is a test-suite-based repair tool for C programs that has been created to compare the performance between genetic programming (i.e., GenProg) and random search in the case of automatic program repair. Qi et al. (2014) showed that, in most cases (23/24), RSRepair finds valid patches faster than GenProg.

**Nopol (XUAN et al., 2016).** Nopol is a semantics-based repair tool dedicated to repairing buggy `if` conditions and to adding missing `if` preconditions. Nopol collects angelic values at runtime to determine the expected behavior of suspicious statements: an angelic value is an arbitrary value that makes all failing test cases from the program under repair pass. Then, those values are encoded into a Satisfiability Modulo Theory (SMT) formula to find an expression that matches the behavior of the angelic values. When the SMT formula is satisfiable, Nopol translates the SMT solution into a source code patch.

**DynaMoth (DURIEUX; MONPERRUS, 2016a).** DynaMoth is a repair tool integrated into Nopol that also targets buggy and missing `if` conditions. The difference between DynaMoth and Nopol is that, instead of using a SMT formula to generate a patch, DynaMoth uses the Java Debug Interface to access the runtime context and collects variable and method calls. Then, DynaMoth combines those variables and method calls to generate more complex expressions until it finds one that has the expected behavior. This allows the generation of patches that contain method calls with parameters, for instance.

**NPEFix (DURIEUX et al., 2017).** NPEFix is different from the generate-and-validate and semantics-based tools: it is a metaprogramming-based tool. It means that NPEFix modifies the program under repair to include several repair strategies that can be activated during the runtime. NPEFix repairs programs that crash due to a null pointer exception. NPEFix runs the failing test case several times and activates a different repair strategy for each execution. In the end, knowing the repair strategies that have worked, together with information about the context in which they worked, a patch is created. Note that NPEFix works in a similar way to semantics-based tools in this last step: if a patch is found, it means that the patch is already satisfactory.

## 2.2 Benchmarks of Bugs for Automatic Repair

Benchmarks of bugs play an important role in software bug-related research fields. According to Le Goues et al. (2015), a well-designed benchmark 1) simplifies experimental reproduction, 2) helps to ensure generality of results, 3) allows direct comparisons between competing methods, and 4) enables measurement of the progress of a research field over time. There are several benchmarks of bugs that have been used in software bug-related research fields to support empirical evaluations. Several benchmarks were first created for the software testing research community, such as Siemens (HUTCHINS et al., 1994) and SIR (DO et al., 2005), two notable and well-cited benchmarks. The majority of bugs in these two benchmarks were seeded in existing program versions without bugs. Other benchmarks were also created for bug detection research, such as BugBench (LU et al., 2005) and iBugs (DALLMEIER; ZIMMERMANN, 2007), which were built by manual effort or do not include a bug-triggering test for all bugs.

The seminal benchmarks of bugs dedicated to automatic program repair studies were for C programs, namely ManyBugs and IntroClass (LE GOUES et al., 2015). ManyBugs contains 185 bugs collected from nine large, popular, open-source programs. On the other hand, IntroClass targets small programs written by novices and contains 998 bugs collected from student-written versions of six small programming assignments in an undergraduate programming course. There is also Codeflaws (TAN et al., 2017) for C, which contains 3,902 bugs extracted from programming contests available on Codeforces.

For the Java language, the benchmarks of bugs for automatic program repair research are presented in Table 2. IntroClassJava (DURIEUX; MONPERRUS, 2016b) and QuixBugs (LIN et al., 2017) contain bugs that were transpiled to the Java language, i.e., the bugs were not found in the wild as they are. Defects4J (JUST et al., 2014) and Bugs.jar (SAHA et al., 2018), on the other hand, contain real bugs from Java programs.

Table 2 – Benchmarks of bugs for automatic program repair studies in Java.

| Benchmark | # Bugs | # Projects |
|---|---|---|
| Defects4J (JUST et al., 2014) | 395 | 6 |
| IntroClassJava (DURIEUX; MONPERRUS, 2016b) | 297 | 6 |
| QuixBugs (LIN et al., 2017) | 40 | 40 |
| Bugs.jar (SAHA et al., 2018) | 1,158 | 8 |

Defects4J (JUST et al., 2014) (395 bugs from six projects) was created for the software testing community but has been extensively used to evaluate state-of-the-art repair tools (see Section 2.3). More recently, Bugs.jar (SAHA et al., 2018) (1,158 bugs from eight projects) was built, increasing the number of Java bugs available for the automatic program repair community. Both Defects4J and Bugs.jar are based on the same construction approach, which consists of going through commits in version control systems by using information provided by bug trackers to find bug-fixing commits.

However, they include bugs from only 13 projects in total (one project is in common), which are all mature ones. This is a major threat because bugs in benchmarks should come from a representative sample of real-world projects considering diversity in several aspects. Moreover, the current benchmarks are rarely updated, if so. For instance, since its creation in 2014, Defects4J has evolved only once, where 38 bugs collected from a single project were included in the benchmark. Since new projects, new development practices, and new language features are proposed over time, new releases of a benchmark are desirable to keep it relevant. To do so, the benchmark must be extensible upfront, with a modular and reusable design.

In addition, there is a lack of knowledge on benchmarks of bugs for Java. We found only one work, which was published in the same year as our first work on benchmark analysis (SOBREIRA et al., 2018). Motwani et al. (2018) annotated each bug in Defects4J with eleven abstract parameters regarding five defect characteristics: defect importance, complexity, independence, test effectiveness, and characteristics of the human-written patch. An example of an abstract parameter is the number of lines edited in a patch, which is used to compute the defect complexity.

## 2.3 Evaluation of Automatic Repair Tools on Benchmarks of Bugs

Automatic repair tools meet benchmarks of bugs when they are evaluated. In this section, we present a review of the literature of the existing evaluations of the 24 repair tools presented in Table 1. This review is focused on the used benchmarks and the number of bugs given as input to the repair tools. There are two types of scientific papers that are interesting for us: the first type consists of the presentation of a new repair approach, which also includes an evaluation conducted using a tool that implements the approach (e.g., Xuan et al. (2016)'s paper); and the second type consists of an empirical evaluation carried out on already created tools, which is a specific work to evaluate repair tools by running them on benchmarks of bugs (e.g., Martinez et al. (2017)'s paper). We gathered 18 papers from the first type of papers (more than one tool can be presented in the same paper) and two papers from the second type.

Table 3 summarizes our review of the existing evaluations of the 24 repair tools based on the 20 scientific papers. Each repair tool is associated with one or more benchmarks used in its evaluation. When a repair tool has been evaluated on more than one benchmark (or more than once on the same benchmark), we first place the benchmark used in the paper that presented the tool (i.e., first evaluation), followed by the other benchmarks with the reference for the posterior studies. For instance, in the paper in which jGenProg (MARTINEZ; MONPERRUS, 2016) is presented, there is an evaluation on Defects4J: this evaluation has no citation in the second column of the table because it is in the paper

Table 3 – Test-suite-based program repair tools for Java and the benchmarks used in their evaluation.

| Repair tool | Benchmark used in evaluation | # Bugs | # Patched[a] / # Fixed[b] |
|---|---|---|---|
| ACS (XIONG et al., 2017) | Defects4J | 224 | 23 / 17 |
| ARJA (YUAN; BANZHAF, 2018) | Defects4J | 224 | 59 / 18 |
| | QuixBugs (YE et al., 2019) | 40 | 4 / 2 |
| CapGen (WEN et al., 2018) | Defects4J | 224 | 25 / 22 |
| | IntroClassJava | 297 | – / 25 |
| Cardumen (MARTINEZ; MONPERRUS, 2018) | Defects4J | 356 | 77 / – |
| | QuixBugs (YE et al., 2019) | 40 | 5 / 3 |
| DeepRepair (WHITE et al., 2019) | Defects4J | 374 | 51 / – |
| DynaMoth (DURIEUX; MONPERRUS, 2016a) | Defects4J | 224 | 27 / – |
| | QuixBugs (YE et al., 2019) | 40 | 2 / 1 |
| Elixir (SAHA et al., 2017) | Defects4J | 82 | 41 / 26 |
| | Bugs.jar | 127 | 39 / 22 |
| GenProg-A (YUAN; BANZHAF, 2018) | Defects4J | 224 | 36 / – |
| HDRepair (LE et al., 2016) | Defects4J | 90 | – / 23 |
| Jaid (CHEN et al., 2017) | Defects4J | 138 | 31 / 25 |
| jGenProg (MARTINEZ; MONPERRUS, 2016) | Defects4J | 224 | 29 / – |
| | Defects4J (MARTINEZ et al., 2017) | 224 | 27 / 5 |
| | QuixBugs (YE et al., 2019) | 40 | 2 / 0 |
| jKali (MARTINEZ; MONPERRUS, 2016) | Defects4J | 224 | 22 / – |
| | Defects4J (MARTINEZ et al., 2017) | 224 | 22 / 1 |
| | QuixBugs (YE et al., 2019) | 40 | 2 / 1 |
| jMutRepair (MARTINEZ; MONPERRUS, 2016) | Defects4J | 224 | 17 / – |
| | QuixBugs (YE et al., 2019) | 40 | 3 / 1 |
| Kali-A (YUAN; BANZHAF, 2018) | Defects4J | 224 | 33 / – |
| LSRepair (LIU et al., 2018b) | Defects4J | 395 | 38 / 19 |
| Nopol (XUAN et al., 2016) | ConditionDataset | 22 | 17 / 13 |
| | Defects4J (MARTINEZ et al., 2017) | 224 | 35 / 5 |
| | QuixBugs (YE et al., 2019) | 40 | 3 / 1 |
| NPEFix (DURIEUX et al., 2017) | NPEDataset | 16 | 14 / – |
| | QuixBugs (YE et al., 2019) | 40 | 2 / 1 |
| PAR (KIM et al., 2013) | PARDataset | 119 | 27 / – |
| RSRepair-A (YUAN; BANZHAF, 2018) | Defects4J | 224 | 44 / – |
| | QuixBugs (YE et al., 2019) | 40 | 4 / 2 |
| SimFix (JIANG et al., 2018) | Defects4J | 357 | 56 / 34 |
| SketchFix (HUA et al., 2018) | Defects4J | 357 | 26 / 19 |
| SOFix (LIU; ZHONG, 2018) | Defects4J | 224 | – / 23 |
| ssFix (XIN; REISS, 2017b) | Defects4J | 357 | 60 / 20 |
| xPAR (LE et al., 2016) | Defects4J | 90 | – / 4 |

[a] A *patched* bug means that a repair tool fixed it with a test-suite adequate patch.
[b] A *fixed* bug means that a repair tool fixed it with a test-suite adequate patch that was confirmed to be correct.

in which jGenProg is presented. Later, it was again evaluated on Defects4J (MARTINEZ et al., 2017) and also on QuixBugs (YE et al., 2019), which contain citations of the empirical evaluation papers in the table. The table also presents the number of bugs given as input to the repair tools, and the number of bugs for which the tools generated at least one test suite adequate patch (i.e., patched bugs) and correct patch (i.e., fixed bugs), reported by the gathered papers.

In total, we found 38 evaluations of the 24 repair tools. Out of 24, 22 repair tools were evaluated on (a subset of) bugs from Defects4J, and nine of them were recently evaluated on QuixBugs. In some exceptional cases, Bugs.jar and IntroClassJava were also used. However, the number of existing evaluations in terms of the number of repair tools versus the number of benchmarks is low compared to all possible combinations. There are some benchmarks that have been rarely or never used so far: this is partially explained by the

fact that some benchmarks were recently published, thus they were not available when some repair tools were published.

We also observe that three repair tools were originally evaluated on datasets that were not presented in the literature in dedicated research papers (i.e., PARDataset (KIM et al., 2013), ConditionDataset (XUAN et al., 2016), and NPEDataset (DURIEUX et al., 2017)). This is the case of the first evaluations of PAR, Nopol, and NPEFix. However, these repair tools were later evaluated on formally proposed benchmarks, except for PAR, which is not publicly available. PAR was later reimplemented, resulting in the tool xPAR (LE et al., 2016), which was then evaluated on Defects4J.

## 2.4 Final Remarks

In this chapter, we presented general concepts on automatic program repair and benchmarks of bugs, followed by a literature review. The literature review is focused on test-suite-based repair tools and benchmarks of bugs for the Java programming language. Through this review, we ground our problem statement presented in Section 1.1.

There are four benchmarks of bugs for Java: Defects4J, IntroClassJava, QuixBugs, and Bugs.jar. Only Defects4J and Bugs.jar contain bugs from real Java projects. However, they have bugs from only 13 projects, collected with the same strategy. In our literature review, we also found only one work that presents a benchmark characterization: the benchmark considered in that study is Defects4J. Finally, we presented a mapping of repair tools and the benchmarks used for their evaluation, and we found that Defects4J has been extensively used.

# Chapter 3

# Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies

Benchmarks of bugs are essential to empirically evaluate automatic program repair tools. In 2015, Le Goues et al. (2015) claimed that "*Since 2009, research in automated program repair [...] has grown to the point that it would benefit from carefully constructed benchmarks*", which motivates research towards the construction of well-designed benchmarks of bugs. A well-designed benchmark 1) simplifies experimental reproduction, 2) helps to ensure generality of results, 3) allows direct comparisons between competing methods, and 4) enables measurement of a research field progress over time (LE GOUES et al., 2015). Collecting bugs, however, is a challenging task. Le Goues et al. (2013) highlighted that a key challenge they faced was finding a good benchmark of bugs to evaluate their repair tool. They also claimed that a good benchmark should include *real*, *reproducible* bugs from a variety of real-world systems.

To address the scarcity of benchmarks of bugs, we present Bears, a project to collect and store bugs in an extensible bug benchmark for automatic repair studies in Java. Bears is divided into two components: the Bears-collector and the Bears-benchmark. The former is the tool for collecting and storing bugs, and the latter is the actual benchmark of bugs. Unlike Defects4J and Bugs.jar, the approach employed by the Bears-collector to automatically go through past commits in the history of projects is based on commit building state from Continuous Integration (CI) builds. Our idea is that the statuses of CI builds indicate compilable and testable program versions, which are the first requirements for a reproducible bug and its patch. Moreover, Bears uses GitHub in an original way in order to support the benchmark evolution. The Bears-collector automatically creates a publicly available branch in a given GitHub repository for each successfully reproduced bug. One can use these branches to extend Bears by opening pull requests in the main Bears-benchmark repository.

This chapter is organized as follows. Section 3.1 presents the design decisions in the conception of BEARS. Section 3.2 presents the process of the BEARS-COLLECTOR, which is used in three execution rounds to collect bugs for the first version of the BEARS-BENCHMARK in Section 3.3. Section 3.4 presents discussions on challenges, limitations, threats to validity, and related work. Finally, Section 3.5 presents the final remarks.

## 3.1 BEARS Design Decisions

In this section, we present the main design decisions for BEARS: the intuition, the bug identification strategy based on CI build statuses, the criteria that projects must meet to be used as source for collecting bugs with the BEARS-COLLECTOR, the criteria that bugs must meet to be included in the BEARS-BENCHMARK, and the bug repository design.

### 3.1.1 Bug Collection based on Continuous Integration

The closest related works to BEARS, which are Defects4J and Bugs.jar, are based on mining past commits with the support of bug trackers. We explore an original path, which consists of using commit building state from Continuous Integration builds to create our benchmark. Our idea is that the statuses of CI builds (failed, errored, and passed) can guide us in finding compilable and testable program versions that contain test failures and their patches written by developers.

A CI platform that tightly integrates with GitHub is Travis. It has emerged as the most used CI platform for open-source software development, and it has recently started to attract the attention of researchers as a data source to conduct research: the mining challenge at MSR '17 (The 14th International Conference on Mining Software Repositories) was on a dataset synthesized from Travis CI and GitHub (BELLER et al., 2017). In Travis CI, a build marked as *errored* means that some failure has happened in a phase of the build life cycle that is before the execution of tests, thus the version of the program when such build was triggered is not interesting for us (we focus on test failures). On the other hand, builds marked as *passed* suggest compilable and testable program versions.

### 3.1.2 Buggy and Patched Program Versions from CI Builds

In CI, each build $b$ is associated with a program version by its commit[3]. Our approach identifies pairs of buggy and patched program versions from pairs of immediately subsequent CI builds. A given pair ($b_n$, $b_{n+1}$) is referred to as ($b_{buggy}$, $b_{patched}$) when it is associated with a pair of buggy and patched program versions. There are two cases where this can be identified based on the CI build statuses and the files changed between the program versions:

[3] In this chapter, we refer to a specific CI build by its ID with a hyperlink for its visualization in Travis. Latest access to these links: December 13, 2018.

**Case #1: failing-passing builds with no test change.**

*Definition:* $b_n$ is a failing build and $b_{n+1}$ is a passing build that does not contain changes in test files, and $b_n$ fails because at least one test case fails.

*Example:* Consider the pair of builds ($b_n$ = 330246430, $b_{n+1}$ = 330267605) illustrated in Figure 5. $b_n$ failed in Travis CI because one test case failed due to `ComparisonFailure`, and $b_{n+1}$ passed, where the message of the commit associated with it is "fix test".

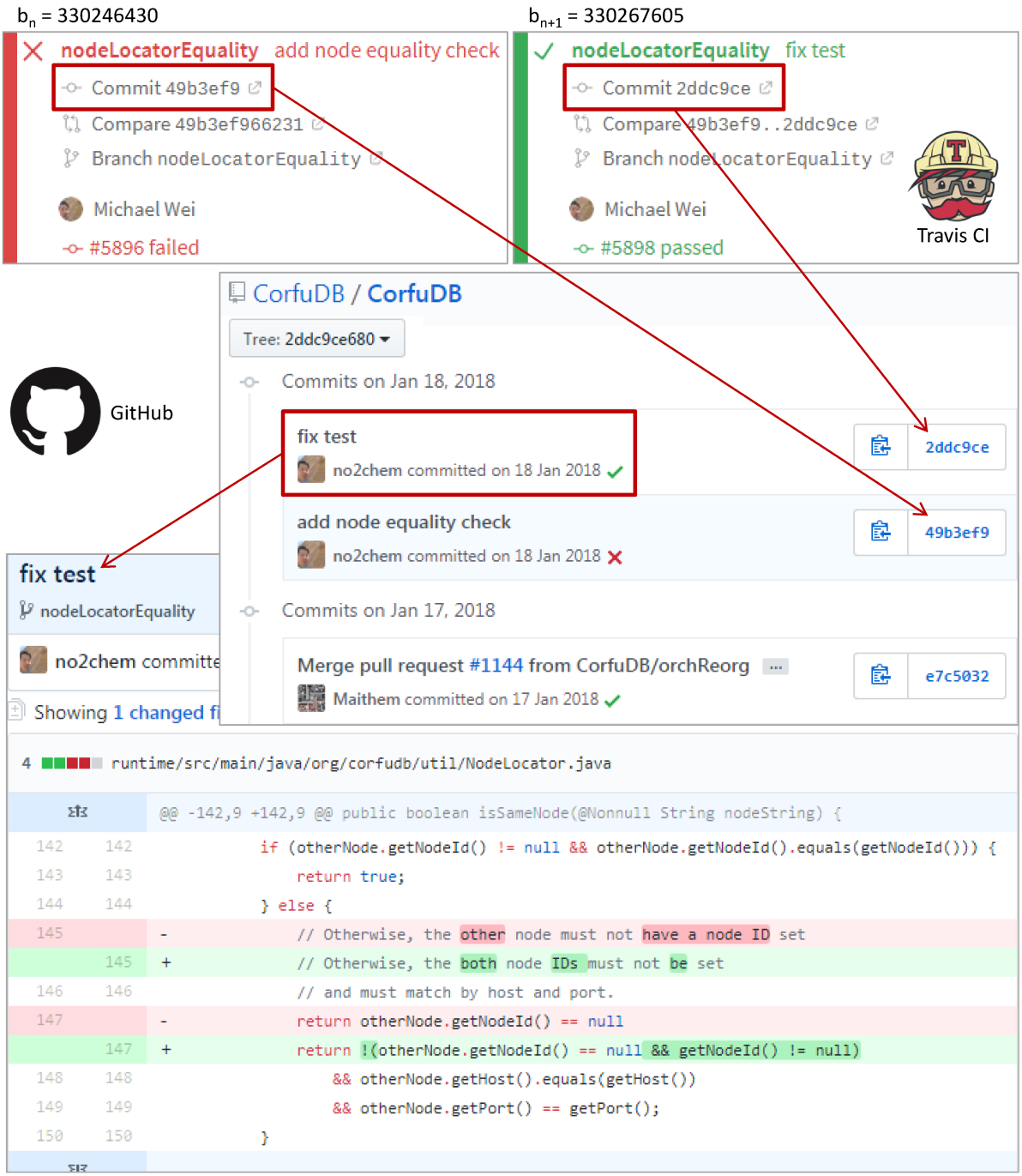


Figure 5 – Example of case #1: failing-passing builds with no test change.

**Case #2: passing-passing builds with test changes.**

*Definition:* $b_n$ is a passing build and $b_{n+1}$ is also a passing build, but $b_{n+1}$ contains changes in test files, and at least one test case fails when the source code from $b_n$ is tested with the test cases from $b_{n+1}$.

*Example:* Consider the pair of builds ($b_n$ = 330973656, $b_{n+1}$ = 330980388) illustrated in Figure 6. Both builds passed in Travis CI. However, a test file was changed in the commit that triggered $b_{n+1}$. The source code from $b_n$, when tested with the tests from $b_{n+1}$, fails due to `NullPointerException`. Indeed, the commit message from $b_{n+1}$, "Fix NPE for GCS buckets that have underscores", confirms that a null pointer exception had been fixed.

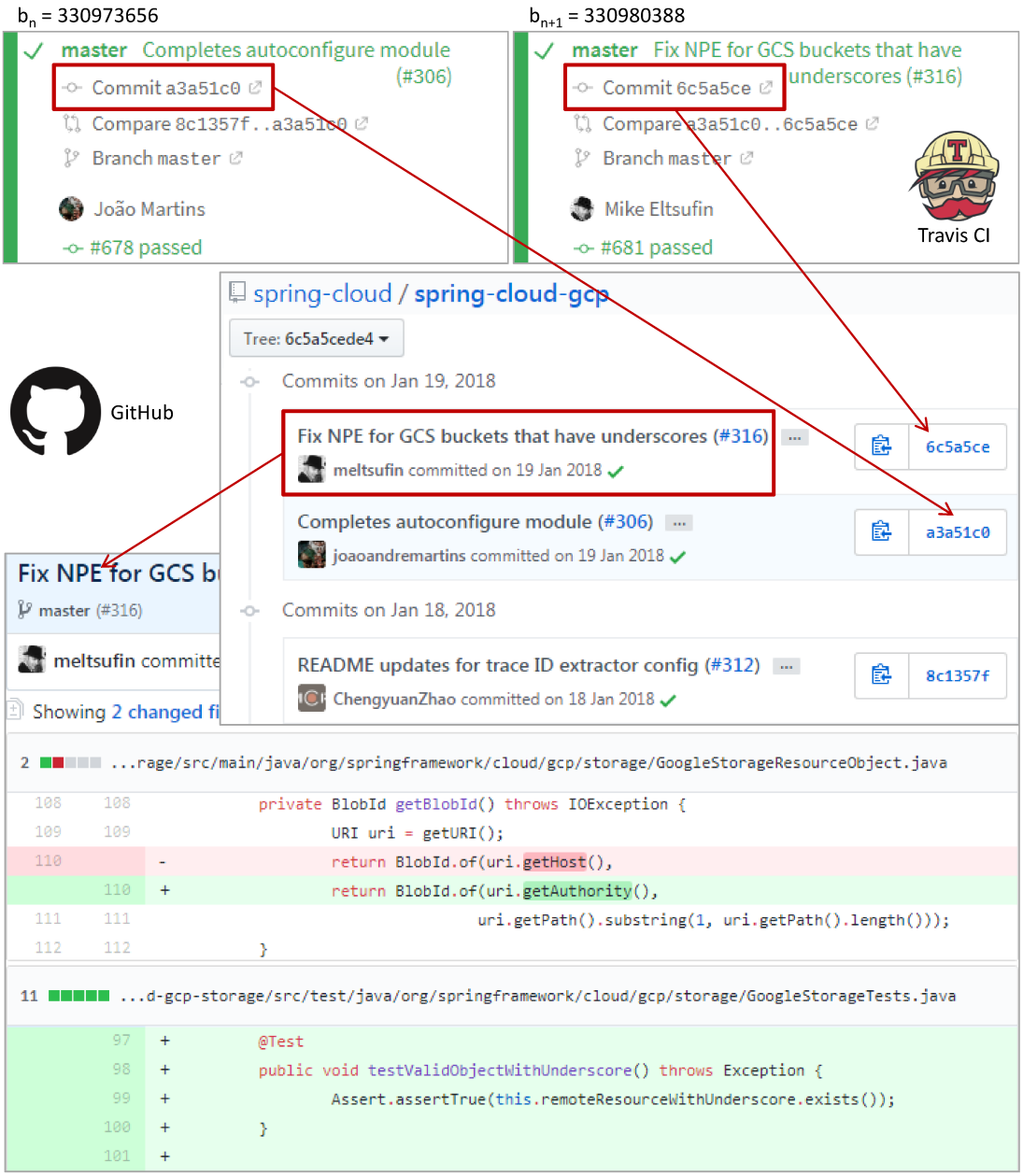


Figure 6 – Example of case #2: passing-passing builds with test changes.

The difference between both cases is the bug-triggering test. In case #1, the test is contained in $b_n$ ($b_{buggy}$), which makes $b_n$ fail in Travis. In case #2, the test is contained in $b_{n+1}$ ($b_{patched}$) together with the patch, which makes $b_n$ pass in Travis since there is no bug-triggering test on it.

### 3.1.3 Inclusion Criteria for Projects

The criteria a project must meet to be considered by the BEARS-COLLECTOR are the following: 1) it must be publicly available on GitHub, 2) it must use the Travis Continuous Integration service, and 3) it must be a Maven project. These three conditions are required to keep engineering effort feasible in a research context (supporting different code hosting, CI services, and build technologies, although possible, are out of scope).

### 3.1.4 Inclusion Criteria for Bugs

The criteria a bug must meet to be included in the BEARS-BENCHMARK are the following:

*Criterion #1–The bug must be reproducible.* To repair a bug in a given program, test-suite-based repair tools rely on the test suite of the program containing at least one failing test case, so that they generate patches that make the whole test suite pass. Therefore, each bug contained in the BEARS-BENCHMARK must be accompanied with at least one bug-triggering test case.

*Criterion #2–The bug must have been fixed by a human.* A patch generated by a test-suite-based repair tool, even when it makes the whole test suite pass, may be incorrect. This is because the patch might overfit the test cases and not generalize to all inputs. When manually evaluating the correctness of a patch generated by a repair tool for a given bug, one of the most valuable resources that researchers use is the human-written patch. To allow this type of study, which is essential for making sound progress, we ensure that each bug contained in the BEARS-BENCHMARK is accompanied with its human-written patch.

### 3.1.5 Bug Repository Design

A well-designed bug collection process involves the important feature of storing the bugs and their patches. We designed such a feature aiming 1) to keep bugs organized, 2) to make bugs publicly available for the research community, and 3) to make it easier for other researchers to collect more bugs using the BEARS-COLLECTOR and include them in the BEARS-BENCHMARK.

To achieve goal #2, we decided to automatically store bugs in a public GitHub repository, which is given as input to the BEARS-COLLECTOR. To achieve goal #1, we defined

Table 4 – The canonical commits in a BEARS git branch.

| Commit # | Cases | Commit content |
|---|---|---|
| Commit #1 | both | the version of the program with the bug |
| Commit #2 | case #2 | the changes in the tests |
| Commit #3 | both | the version of the program with the human-written patch |
| Commit #4 | both | the metadata file `bears.json` |

that the internal organization of such a repository is based on branches, so that when a pair of builds is successfully reproduced by the BEARS-COLLECTOR, a new branch is created in the given GitHub repository. Since the repository is settable in the BEARS-COLLECTOR, and the result produced by the BEARS-COLLECTOR is branch-based, we achieve goal #3 by allowing pull requests in the BEARS-BENCHMARK repository from the given GitHub repository (when it is a fork from the BEARS-BENCHMARK repository).

The branches generated by the BEARS-COLLECTOR are standardized. The name of each branch follows the pattern `<GitHub project slug>-<buggy build id>-<patched build id>`, and each branch contains the sequence of commits presented in Table 4. In addition, each branch contains a special file named `bears.json`, which is a gathering of information collected during the bug reproduction process. This file is based on a `json` schema that we created to store key properties. It contains information about the bug (e.g., test failure names), the patch (e.g., patch size), and the bug reproduction process (e.g., duration).

## 3.2 The BEARS-COLLECTOR Process

The overview of the BEARS process for collecting and storing bugs is illustrated in Figure 7. In a nutshell, given an input configuration, pairs of builds are scanned from Travis CI (*Build Scanning*). Each pair of builds passes through a reproduction pipeline towards finding a test failure followed by a passing test suite, i.e., a reproducible bug and its patch (*Reproduction*). Each successfully reproduced pair is stored in a dedicated branch in a GitHub repository. If such repository is a fork from the BEARS-BENCHMARK repository, its branches can be used for opening pull requests for the addition of bugs into the BEARS-BENCHMARK: the special branch named "pr-add-bug" is used as the base branch. An open pull request is automatically validated by a build triggered in Travis CI and manually validated by a collaborator of BEARS (*Validation*). If the pull request passes in both validations, it is merged, and a new branch for the bug is created in the BEARS-BENCHMARK; otherwise, it is closed. The three main phases of this process are presented in dedicated sections as follows.

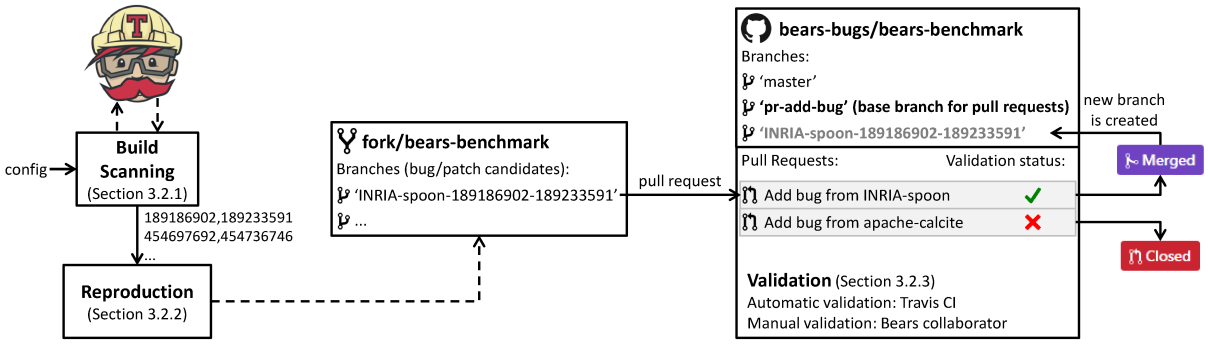


Figure 7 – Overview of the BEARS process: its uniqueness is to be based on commit building state from Travis CI.

### 3.2.1 Phase I–Build Scanning

The first phase of the BEARS-collector process is the build scanning. The goal of this phase is to automatically scan build IDs from Travis CI and select pairs of them to be reproduced (Phase II–Reproduction), such that the selected pairs correspond to one of the two cases presented in Section 3.1.2.

The scanning can be performed in two ways: 1) for a fixed period or 2) in real time. In the former, the scanner takes as input a list of projects (that meet the criteria presented in Section 3.1.3) and a time window, and scans all builds from the projects that have finished to run in Travis during such a time window. In the latter, the scanner repeatedly checks Travis (e.g., in each minute) for builds that have finished running independently of a given list of projects.

In both ways, after scanning build IDs, pairs of build IDs are created, and then they pass through a selection process. To do so, we first gather the builds that passed in Travis out of all scanned builds, which are the $b_{patched}$ candidates. Then, each $b_{patched}$ candidate is given as input to Algorithm 1, where its previous build is retrieved (line 1), the $b_{buggy}$ candidate.

The algorithm checks if the $b_{buggy}$ candidate failed in Travis (line 4). If so, and if the $diff$ (changes) between the commits that triggered the two builds contains Java source code files and does not contain test files (line 5), the pair ($b_{buggy}$, $b_{patched}$) is successfully returned (line 6) because it can fit in case #1 (failing-passing builds with no test change). If the $b_{buggy}$ candidate passed in Travis (line 8), and if the $diff$ between the commits that triggered the two builds contains Java source code files and also test files (line 9), the pair ($b_{buggy}$, $b_{patched}$) is successfully returned (line 10) because it can fit in case #2 (passing-passing builds with test changes). Each pair of builds collected with this algorithm is given as input to Phase II–Reproduction.

Algorithm 1 – Assessing suitability of a scanned $b_{patched}$ candidate for reproduction.

**Input:** $b_{patched}$ candidate
**Output:** pair ($b_{buggy}$, $b_{patched}$) or *null*
1: $b_{buggy}$ candidate ← previous build from $b_{patched}$ candidate
2: $c_{buggy}$ ← commit that triggered $b_{buggy}$
3: $c_{patched}$ ← commit that triggered $b_{patched}$
4: **if** $b_{buggy}$ is failing in Travis CI **then** ▷ case #1
5: **if** failure in Travis is due to test failure **and**
$diff(c_{buggy}, c_{patched})$ contains Java source code file **and**
$diff(c_{buggy}, c_{patched})$ does not contain Java test file **then**
6: **return** ($b_{buggy}$, $b_{patched}$)
7: **end if**
8: **else** ▷ case #2
9: **if** $diff(c_{buggy}, c_{patched})$ contains Java source code file **and**
$diff(c_{buggy}, c_{patched})$ contains Java test file **then**
10: **return** ($b_{buggy}$, $b_{patched}$)
11: **end if**
12: **end if**
13: **return** *null*

### 3.2.2 Phase II–Reproduction

The goal of this phase is to test if pairs of builds obtained in Phase I–Build Scanning fit in one of the two cases presented in Section 3.1.2. To do so, we submit each pair of builds to a *reproduction* attempt process, which builds both program versions from the pair of builds, runs tests, and analyzes the test results. If at least one test failure is found in the program version from the $b_{buggy}$ candidate, but not in the one from the $b_{patched}$ candidate, a bug and its patch were reproduced.

To perform the reproduction, we designed and implemented a *pipeline of steps*. Algorithm 2 presents the core steps for a given build pair candidate ($b_{buggy}$, $b_{patched}$). This algorithm has two main phases: the pipeline construction (lines 1 to 20) and the pipeline execution (lines 21 to 28). The pipeline construction consists of creating a sequence of steps that should be further executed in order, which are stored in the variable *pipeline* (line 1). The pipeline execution consists of running the *pipeline* step by step. If any step fails, the execution of the pipeline is aborted (line 25). Otherwise, the execution of the pipeline ends with success (line 28), meaning that a test failure was found in the program version from the $b_{buggy}$ candidate and no test failure was found in the program version from the $b_{patched}$ candidate. Note that some pipeline steps are responsible for storing data by committing and pushing to a GitHub repository, according to the design for storing bugs described in Section 3.1.5. These steps are in Algorithm 3 and Algorithm 4, which are called from the main reproduction algorithm (Algorithm 2) in lines 14 and 20. We separated these steps from the main algorithm to make it easier to understand the core steps of the reproduction process presented in Algorithm 2, but yet providing the completeness of the process.

Algorithm 2 – Assessing reproducibility for a build pair.

**Input:** pair ($b_{buggy}$, $b_{patched}$)
**Input:** *repo*: a GitHub repository where a new branch is created if the reproduction attempt succeeds
**Output:** reproduction attempt status (it indicates success or failure of the reproduction attempt, and in case of success, a branch has been created in *repo* by the algorithm)

**var** *pipeline*: a queue with FIFO steps
*pipeline* ← clone repository ▷ '←' means 'append'
▷ **Beginning of the reproduction towards test failure(s)**
**if** $b_{buggy}$ is failing in Travis CI **then** ▷ case #1
    *pipeline* ← check out the commit from $b_{buggy}$
**else** ▷ case #2
    *pipeline* ← check out the commit from $b_{patched}$
    *pipeline* ← search source code and test code directories
    *pipeline* ← check out only source code files from $b_{buggy}$
**end if**
*pipeline* ← build project
*pipeline* ← run tests
*pipeline* ← analyze test results ▷ **test failure must be found**
InitRepository(*pipeline*, $b_{buggy}$, $b_{patched}$)
▷ **Beginning of the reproduction towards a passing test suite**
*pipeline* ← check out the commit from $b_{patched}$
*pipeline* ← build project
*pipeline* ← run tests
*pipeline* ← analyze test results ▷ **no test failure must be found**
CreateBranchInRepository(*pipeline*, *repo*)
**for** $i \leftarrow 1$ **to** *pipeline.size* **do**
    *step* ← *pipeline*[*i*]
    *status* ← run *step*
    **if** *status* is failed **then**
        **return** failure status ▷ abort pipeline execution
    **end if**
**end for**
**return** success status

Algorithm 3 – InitRepository(*pipeline*, $b_{buggy}$, $b_{patched}$)

**if** $b_{buggy}$ is failing in Travis CI **then** ▷ case #1
    *pipeline* ← commit files (commit #1 in Table 4)
**else** ▷ case #2
    *pipeline* ← check out only test code files from $b_{buggy}$
    *pipeline* ← commit files (commit #1 in Table 4)
    *pipeline* ← check out only test code files from $b_{patched}$
    *pipeline* ← commit files (commit #2 in Table 4)
**end if**

Algorithm 4 – CreateBranchInRepository(*pipeline*, *repo*)

*pipeline* ← commit files (commit #3 in Table 4)
*pipeline* ← commit files (commit #4 in Table 4)
*pipeline* ← create a new branch and push it to *repo*

The general idea of Algorithm 2 is to try to reproduce a test failure from the source code of the $b_{buggy}$ candidate first, and then a passing test suite from the source code of the $b_{patched}$ candidate. To start, the algorithm makes a local copy of the remote GitHub repository from which the pair of builds were triggered (line 2). Then, the algorithm checks out the repository to the point of interest in the project history. If the $b_{buggy}$ candidate failed in Travis (i.e., the pair of builds is in case #1), the algorithm checks out the commit that triggered $b_{buggy}$ (line 5) in order to test the source code from $b_{buggy}$ with the tests from $b_{buggy}$. If the $b_{buggy}$ candidate passed in Travis (i.e., the pair of builds is in case #2), the algorithm first checks out the commit that triggered $b_{patched}$ (line 7), which contains the tests to be executed in the source code from $b_{buggy}$. Then, the algorithm searches the source code and test code directories (line 8) so that only the source code files from the commit that triggered $b_{buggy}$ can be checked out (line 9) in order to test the source code from $b_{buggy}$ with the tests from $b_{patched}$.

After the checking out steps, the algorithm builds the project (line 11), runs tests (line 12), and analyzes the test results (line 13). Since the $b_{buggy}$ candidate is being tested, at least one test case must fail. If it does happen, it means that the algorithm found a reproducible bug, and its execution continues in order to try to reproduce a passing test suite from the $b_{patched}$ candidate. Then, it checks out the commit that triggered $b_{patched}$ (line 16), builds the project (line 17), runs tests (line 18), and analyzes the test results (line 19). Since the $b_{patched}$ candidate is being tested, all tests must pass. If it does happen, it means that the algorithm reproduced the patch for the bug. At the end of a successful reproduction attempt, a branch is created in a GitHub repository given as input to the algorithm, according to the design presented in Section 3.1.5.

### 3.2.3 Phase III–Validation

To include a branch created in Phase II–Reproduction in the BEARS-BENCHMARK repository, a pull request must be created by using the special branch "pr-add-bug" as the base branch (see Figure 7). Once a pull request is open, it is validated in two steps:

*Automated validation.* The creation of the pull request triggers a build in Travis, which runs a set of scripts to check if the content in the pull request (from the branch) is valid. For instance, one script checks if the buggy program version stored in the branch (commit #1 in Table 4) has at least one failing test case when its test suite is executed.

*Manual validation.* A collaborator of BEARS performs a manual analysis of the pull request to check if the proposed branch contains a genuine bug. It might include, for instance, the analysis of the buggy and patched source code diff and the test failures.

A pull request that passes in both validations is merged, and a new branch is automatically created in the BEARS-BENCHMARK repository with the content of the pull request; otherwise, it is closed.

### 3.2.4 Implementation

The main tools used in the implementation of the Bears-collector are the following. In Phase I–Scanning, we rely on jtravis[4], a Java API to use the Travis CI API. In Phase II–Reproduction, we use different tools depending on the pipeline step. The step *clone repository* and the *check out* ones are performed using JGit[5] and also by directly invoking `git` commands. Since we work with Maven projects, we use the Apache Maven Invoker[6] to invoke Maven in the steps *build project* and *run tests*, and we use the Maven Surefire Report Plugin[7] to gather test information in the step *analyze test results*. To execute the entire pipeline, we use Docker containers, which are based on a Docker image configured to use JDK-8 and Maven 3.3.9. Finally, in Phase III–Validation, we rely on Travis to run the automatic validation.

## 3.3 The Creation of Bears-benchmark

To create the first version of the Bears-benchmark, we used the Bears-collector process described in the previous section to collect bugs. We performed three execution rounds: two rounds are from a given time window and a list of projects, and the third one is from real-time scanning of builds. In this section, we present and discuss our results organized along the three phases of the bug collection process.

### 3.3.1 Scanning 168,772 Travis Builds

In the scanning phase, the Bears-collector scans and selects pairs of builds from Travis CI (see Section 3.2.1), which are associated with pairs of program versions to be submitted to the reproduction attempt process. We used the following input configuration for each execution round:

*Execution round #1 (time window Jan.–Dec. 2017).* We first queried the most popular Java projects based on the number of watchers on GHTorrent (gousios, 2013). Then, we filtered out the projects that do not meet the criteria presented in Section 3.1.3, remaining ∼1,600 projects. These projects were used in early experimentation during the development of the Bears-collector. We then selected four of them based on the number of successfully reproduced buggy and patched program versions in the early experimentation: INRIA/spoon, traccar/traccar, FasterXML/jackson-databind, and spring-

[4] jtravis: <https://github.com/Spirals-Team/jtravis>. Latest access: December 13, 2018.
[5] JGit: <https://github.com/eclipse/jgit>. Latest access: December 13, 2018.
[6] Apache Maven Invoker: <https://maven.apache.org/shared/maven-invoker>. Latest access: December 13, 2018.
[7] Maven Surefire Report Plugin: <http://maven.apache.org/surefire/maven-surefire-report-plugin>. Latest access: December 13, 2018.

projects/spring-data-commons. We set up the scanning of the execution round #1 with these four projects to comprehensively scan all builds from January to December 2017.

*Execution round #2 (time window Jan.–Apr. 2018).* We queried the top-100 projects that could be reproduced in the Repairnator project (URLI et al., 2018), which were used for the scanning of the execution round #2 from January to April 2018.

*Execution round #3 (real time).* We also scanned pairs of builds in real time. By doing so, there is no need to give a list of projects as input to the BEARS-COLLECTOR, since the scanner directly fetches builds from Travis in real time. We collected data with the execution round #3 during ∼2 months in 2018 (September to November).

Table 5 summarizes the input and the scanning results per execution round. In total, we scanned 168,772 builds over one year and a half. From these scanned builds, we obtained 12,355 pairs of builds from Algorithm 1, where 741 pairs are in case #1 (failing-passing builds) and 11,614 pairs are in case #2 (passing-passing builds).

Table 5 – Scanning results over three rounds.

| | Execution Round | | | Total |
|---|---|---|---|---|
| | #1 | #2 | #3 | |
| Period | 1 year | 4 months | ∼2 months | |
| # Input projects | 4 | 100 | – | |
| # Total scanned builds | 4,987 | 66,621 | 97,164 | 168,772 |
| # Build pairs in case #1 | 17 | 590 | 134 | 741 |
| # Build pairs in case #2 | 1,027 | 7,755 | 2,832 | 11,614 |
| # Total build pairs | 1,044 | 8,345 | 2,966 | 12,355 |

There is a large difference between the number of pairs in case #1 (failing-passing builds) and the number of pairs in case #2 (passing-passing builds): only 6% of the total pairs of builds are in case #1. This suggests that developers do not usually break builds by test failure, or if they do, they fix the test failure in the next build by also changing the test code. Note that, in Algorithm 1, we only accept pairs in case #1 that do not contain test changes between the builds. Our idea is that the test failure that happened in Travis is fixed in the next build by only changing the source code. Moreover, since both builds passed in Travis in case #2, but there is a test change between them, this case potentially includes pairs where feature addition and refactoring are performed, which explains the higher number of pairs in case #2 over case #1.

### 3.3.2 Reproducing 12,355 Pairs of Builds

The 12,355 pairs of builds obtained in the scanning phase were all submitted to the reproduction pipeline presented in Algorithm 2 (see Section 3.2.2). Each reproduction attempt ends with a status that indicates either the success or failure of the reproduction

itself. Figure 8 shows the distribution of the 12,355 reproduction attempts per status. We had 856 successful reproductions that resulted in branches, which is 7% of the reproduction attempts. We report on the failure statuses of the 93% reproduction attempts that failed as follows.

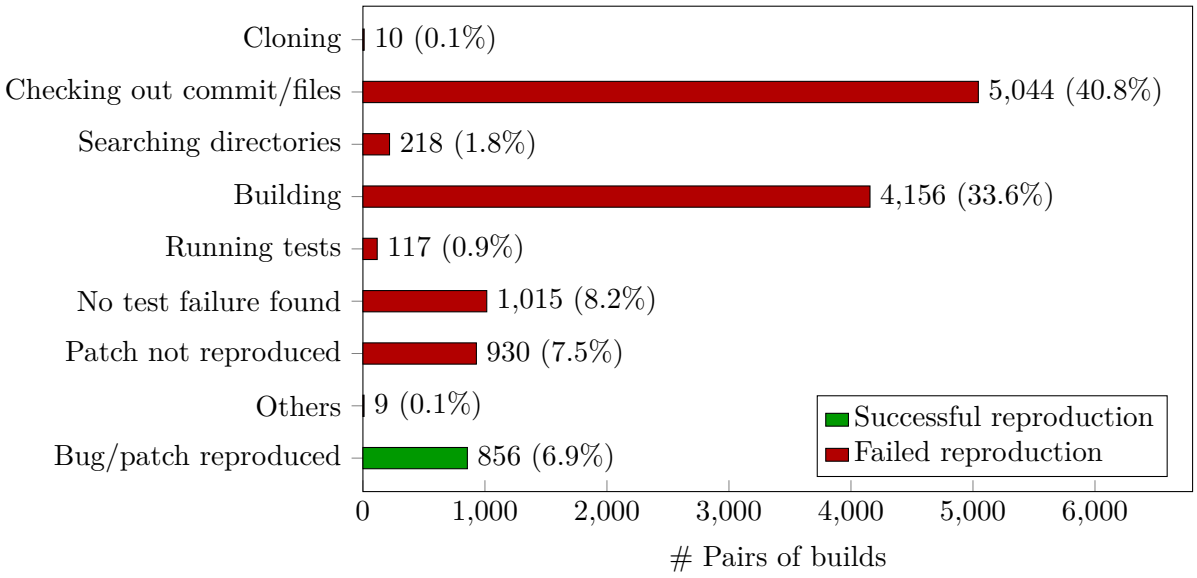


Figure 8 – Statuses of the reproduction attempts.

*Failure when cloning.* In the execution round #3 (real time), we had ten reproduction attempts that failed in the first reproduction pipeline step, i.e., when cloning the remote GitHub repository for a pair of builds. All of these occurrences happened with pairs of builds from the same project.

*Failure when checking out.* The most frequent failure in the reproduction attempts, occurring in 40.8% of all attempts, is due to the checkout of commits. This failure occurs when the commit is missing from the history of the GitHub repository, but the build triggered by it remains in Travis. A missing commit occurs when it was directly deleted or its corresponding branch was deleted. The former might happen in several ways with advanced usage of Git (e.g., with the 'git commit –amend' and 'git reset' commands). The latter might happen on branches used in merged pull requests.

*Failure when searching directories.* For pairs of builds in case #2, there is a pipeline step that searches for the directories containing source code and test code. In a few cases (1.8%), this step did not succeed in finding the two types of directories.

*Failure when building.* 33.6% of the reproduction attempts failed when building (including compiling) the project. This is due to three main reasons. First, some dependencies were missing. This happens, for instance, when dependencies are not declared in the `pom.xml` file of the project. Second, due to case #2, when the source code and test code files from two different commits are mixed, compilation errors naturally occur. Third, we set up a

timeout for each Maven goal used in the pipeline: if no output is received from Maven in ten minutes (same value as Travis CI), the pipeline is aborted.

*Failure when running tests.* In a few cases (0.9%), reproduction attempts failed during the execution of tests. We observed that this is mainly due to the timeout on Maven goal.

*No test failure found.* The first program version (from the $b_{buggy}$ candidate) is supposed to result in at least one test failure when the test suite is executed. In 8.2% of the reproduction attempts, the whole test suite passed, so the reproduction was aborted. This naturally happens in case #2, since both builds passed in Travis, which means that the changes in the tests between the two program versions do not trigger any bug. In case #1, this can be explained, for instance, by flaky tests (LUO et al., 2014), with test failures happening in Travis but not locally.

*Patch not reproduced.* The second program version (from the $b_{patched}$ candidate) is supposed to result in a whole passing test suite. However, this did not occur in 7.5% of the reproduction attempts. As in *no test failure found* status, flaky tests could also be one of the reasons why test failures occurred locally but not in Travis. Moreover, the environment used locally might be different from the one used in Travis (e.g., Java version), leading to different test results.

*Other issues.* Nine reproduction attempts failed due to other five different reasons. First, we had three reproduction attempts that stayed running for up to three hours, constantly generating output, so the timeout with no output was not reached. We forced these reproductions to stop. Second, two reproduction attempts crashed in committing data steps (see Algorithm 3 and Algorithm 4) due to lack of disk space. Third, two reproduction attempts did not succeed in pushing a branch (the last step of the pipeline). Fourth, one reproduction attempt was aborted in the very beginning (before cloning the repository) due to a network issue. Finally, one reproduction attempt crashed due to an uncaught exception when gathering test information with the Maven Surefire Report Plugin.

### 3.3.3 Validating 856 Branches

The 856 branches generated in the reproduction phase passed through the two-step validation phase (see Section 3.2.3). The execution order of the automatic and manual validations does not change the result: only branches that are considered valid in both validations are included in the BEARS-BENCHMARK. For the first version of the BEARS-BENCHMARK, the author of this thesis started by the manual validation to have a complete view of the obtained branches, and then the manually validated ones were submitted to the automatic validation[8].

[8] For future studies, we suggest the conduction of the automatic validation before the manual validation in order to minimize manual effort.

Out of 856, 295 branches were considered valid in the manual validation, and 251 of them were successfully validated by the automatic validation. In the remainder of this section, we report two interesting cases that passed in both validations and the reasons why branches were invalidated.

> *Successful interesting cases*

*The bug-triggering test case already existed in the first passing build in case #2.* Our hypothesis when defining case #2 was that the first passing build contains a bug, but it passed in Travis because the developers did not know about the existence of the bug, or they just did not write a test case to trigger it. However, during the manual validation, we found a few cases where the bug-triggering test case already existed in the first passing build, but it was marked to be ignored when running the test suite. This is the case, for instance, of the pair of builds 352481508–352894244 from the raphw/byte-buddy project.

*The commit message hides a bug fix.* The commit message is sometimes unclear on the changes performed in a commit. We found an interesting case that the BEARS-COLLECTOR reproduced, where several test cases failed due to `NullPointerException` when reproducing the $b_{buggy}$ candidate. The message of the commit that triggered $b_{patched}$ candidate, however, is "Refactor tests". The source code diff between the commits that triggered both builds suggested a patch for `NullPointerException`, so we contacted the developer the same day the bug fix commit occurred (this was possible because this case happened in the real-time execution round). The developer confirmed the existence of the bug fix and that he was in the middle of a bug-hunting. This is the case of the pair of builds 437204853–437571024 from the vitorenesduarte/VCD-java-client project.

> *Invalid branches during the manual validation*

*Refactoring/cleaning.* We obtained several refactorings and cleanups from pairs of builds in case #2 (passing-passing builds). For instance, the pair of builds 330415018–330418847 from the aicis/fresco project is related to cleaning instead of a bug fix. A part of the source code was removed, and the test code was adapted so that the tests pass on the cleaned code, but not on the previous version of the code without the cleaning.

*Feature addition/enhancement.* We also obtained feature additions from pairs of builds in case #2. The changes in the tests in the second passing build are either to test new features or adapted to work properly on the source code containing the new features, thus the changed tests fail when executed on the previous version of the source code without the features. For instance, the pair of builds 217413927–217431352 from the traccar/traccar project is related to a feature addition, where the commit associated with the second build contains the message "Add new Aquila protocol format".

*Duplicate patches.* We obtained several branches with duplicate patches due to the following scenario. Some changes (such as bug fixes) were applied in a branch $X$ in a given

project, and we obtained a BEARS branch from the pair of builds related to the changes performed in the branch $X$. Then, these changes were also applied to a branch $Y$ in the given project, and we obtained a BEARS branch from a pair of builds in the branch $Y$ too. As a result, the two BEARS branches happen to contain the same patch. For instance, we obtained branches from the pairs 176912167–190405643 and 183487103–190406891 from the FasterXML/jackson-databind project. The changes performed in the commit that triggered 190405643 in the branch "2.7" were merged into the branch "2.8", creating a new commit and triggering 190406891.

*Unrelated commits combined.* We obtained branches that store patches containing unrelated changes from multiple commits combined due to pairs of builds that are not from immediately subsequent commits. We observed two cases in which this may happen. In the first one, the developer does more than one commit locally and pushes them to the remote repository at once. In this case, only the last pushed commit triggers a build in Travis, meaning that there are commits with no associated build between the build from the last commit and its previous build. In the second case, a build $b_n$ finished running in Travis after the build $b_{n+1}$, so when obtaining the previous build from $b_{n+2}$, we obtain $b_n$. As in the first case, there are commits between the pair of builds. This is the case of the pair of builds 234112974–234114955 from the INRIA/spoon project, where two builds were triggered between the pair of builds.

*Bug fix including other changes.* Finally, we discarded some branches containing bug fixes, but that also included other changes such as refactoring and code formatting. An example of a discarded branch is the one we obtained from the pair of builds 371024842–371501238 from the molgenis/molgenis project. We do not claim that the BEARS-BENCHMARK contains only branches with isolated bug fixes, but we avoided the inclusion of branches containing bug fixes mixed with other changes to make studies on patches, such as the one reported by Sobreira et al. (2018), easier.

*> Invalid branches during the automatic validation*

The automatic validation, which runs checks in Travis on the content of branches (from pull requests in the BEARS-BENCHMARK repository), invalidated branches mainly because a failure occurred when validating the patched program version by building and running tests, which suggests, for instance, the existence of flaky tests.

### 3.3.4 Content of BEARS-BENCHMARK

The collection of bugs presented in the previous section resulted in 251 real, reproducible bugs, which constitute version 1.0 of the BEARS-BENCHMARK. Out of 251, 19 bugs are from builds in case #1 (failing-passing builds with no test change) and 232 bugs are from builds in case #2 (passing-passing builds with test changes).

Table 6 – Excerpt of open-source projects contained in the Bears-benchmark. The diversity of domains, age, and size is higher than Defects4J and Bugs.jar.

| Name | Project Type | Domain | GitHub info[a] Creation | #Contrib. | #Commits | Bears info[b] LOC | #Tests | #Bugs |
|---|---|---|---|---|---|---|---|---|
| INRIA/spoon | Lib | Java code analysis and transformation | 2013/11 | 47 | 2,804 | 76,285 | 1,114 | 62 |
| traccar/traccar | App | GPS tracking system | 2012/04 | 87 | 5,416 | 43,396 | 255 | 42 |
| FasterXML/jackson-databind | Lib | general data binding (e.g. JSON) | 2011/12 | 141 | 5,038 | 99,145 | 1,711 | 26 |
| spring-projects/spring-data-commons | Lib | spring data, data access | 2010/11 | 62 | 1,793 | 36,479 | 2,029 | 15 |
| debezium/debezium | App | platform for change data capture | 2016/01 | 69 | 1,439 | 53,314 | 508 | 7 |
| raphw/byte-buddy | Lib | runtime code generation for the JVM | 2013/11 | 39 | 4,640 | 140,085 | 8,066 | 5 |
| SzFMV2018-Tavasz/AutomatedCar | App | passenger vehicle behavior simulator | 2018/01 | 33 | 862 | 2,196 | 48 | 2 |
| rafonsecad/cash-count | App | accounting software back-end | 2018/09 | 1 | 29 | 753 | 16 | 2 |
| Activiti/Activiti | App | business process management | 2012/09 | 160 | 8,041 | 205,081 | 1,952 | 1 |
| pippo-java/pippo | Lib | micro Java web framework | 2014/10 | 22 | 1,341 | 19,738 | 131 | 1 |

[a] The number of contributors and the number of commits were collected on December 21, 2018.
[b] The metrics LOC and number of tests are averaged over the collected buggy program versions.

The 251 bugs come from 72 projects. Table 6 presents ten out of the 72 projects. The first five projects are the ones with more bugs in the Bears-benchmark, and the other five were randomly selected. For each project, we present its type (library or application), domain, date of creation on GitHub, the number of contributors and commits, and two size metrics (LOC and # Tests) averaged over the collected buggy program versions.

From this table, we note that the bugs in the Bears-benchmark are from at least ten different project domains. Moreover, the projects considerably differ in age and size: there are bugs from an 8-year old project (spring-data-commons) and also from recent projects less than 1-year old (AutomatedCar and cash-count). Considering size, there are bugs from projects ranging from 753 LOC to 205 KLOC, with numbers of tests ranging from 16 to 8K. Overall, this shows that the bugs in the Bears-benchmark are from projects that are diverse in many different aspects.

In the next chapter, which is dedicated to characterizing benchmarks, we present more details on the content of the Bears-benchmark.

## 3.4 Discussion

To date, we are the first to report on a bug collection process based on Continuous Integration. The issues that occurred during the reproduction attempts (e.g., 33.6% of them failed when building) are valuable insights to those interested in researching bug collection. In this section, we present the challenges for creating Bears, its limitations, and the threats to validity.

### 3.4.1 Challenges

The development of the BEARS-collector was itself a challenge. This is mainly due to the high automation required to collect and store bugs. The different steps must be integrated, so that given a configuration for the process, the BEARS-collector scans builds, performs reproduction attempts, and stores the successful ones in a publicly available GitHub repository, with a standard organization and a proper metadata file (`bears.json`) containing information on the reproduction.

A more specific challenge faced is due to case #2 (passing-passing builds with test changes) from multi-module projects and non-standard single-module projects. In case #2, the source code and test code files must be identified, so that they can be mixed for testing the source code of the first passing build with the tests from the second passing build. In a standard single-module project, the source code files are maintained in the folder `src/main/java` and the test files are maintained in the folder `src/test/java`, both folders located in the root folder of the project. However, in multi-module projects, each module has its own organization, and in non-standard single-module projects, the paths of the folders are different from the standard ones. Thus, for case #2, the checkout of source code and test code folders was a concern. To overcome this, given a multi-module project or non-standard single-module project, we search all folders of the project for `pom.xml` files, and we try to compute the source code and test code folders throughout the given paths.

### 3.4.2 Limitations

The BEARS-collector only covers Maven projects: the core steps of the reproduction process were implemented specifically for Maven projects. These steps, in fact, rely on Maven to build projects and run tests. However, the approach itself is independent of the build tool used. Additional work is needed to be able to collect bugs from Travis CI builds that are not from Maven projects (e.g., from Gradle projects).

The environment used in the reproduction process by the BEARS-collector to build and run tests on program versions is a different environment from Travis CI. The environment used by Travis can be configured by the developers, ranging from OS to customized scripts. The environment used by the BEARS-collector is the same for every project. For that reason, the BEARS-collector might fail in reproduction attempts, as well as produce false positive bugs.

An open challenge is the manual validation. Like the state-of-the-art Java benchmarks, a manual analysis is always done before adding a given bug in the benchmark. The manual analysis is a difficult and time-consuming task. Some candidates are simple to validate, for instance, when a supposed bug fixing commit contains a reference to an issue in the repository of the project, describing the issue addressed in the commit. However, other

candidates are harder to validate since the analysis of the source code diff of the two program versions is required. This manual step is a bottleneck to scaling up the process.

### 3.4.3 Threats to Validity

As with any tool, the Bears-collector is not free of bugs. A bug in the Bears-collector could affect the results reported on the execution rounds we performed (presented in Section 3.3). However, the Bears-collector is open-source and publicly available to other researchers and potential users.

Bug candidates may have been incorrectly validated before their inclusion in the Bears-benchmark. For this first version of the Bears-benchmark, the manual analysis (presented in Section 3.3.3) was performed by the author of this thesis. Despite her careful manual analysis, any manual work is prone to mistakes. Our intention is, however, to create an open environment for contributions, where one might 1) flag possibly incorrect branches added in the Bears-benchmark and 2) participate in the manual validation when pull requests are opened in the Bears-benchmark.

Additionally, bugs triggered by flaky tests might exist in the Bears-benchmark. However, each bug was reproduced twice in the same environment (i.e., in the reproduction and automatic validation phases), which mitigates the threat of having bugs with flaky tests.

### 3.4.4 Related Work

The IntroClassJava (durieux; monperrus, 2016b) and QuixBugs (lin et al., 2017) benchmarks contain transpiled buggy program versions from other languages to Java. These benchmarks are far from Bears, which targets real bugs mined in the wild.

The closest benchmarks to Bears are Defects4J (just et al., 2014) and Bugs.jar (saha et al., 2018), both for Java. Defects4J contains 395 reproducible bugs collected from six projects, and Bugs.jar contains 1,158 reproducible bugs collected from eight Apache projects. To collect bugs, the approach used for both benchmarks relies on information from bug tracking systems, and they contain bugs from large, mature projects. Bears, on the other hand, was designed to collect bugs from a diversity of projects other than large and mature ones: we break the need of projects using bug tracking systems. Note that bug tracking systems are used in the direction of documenting bugs. Continuous Integration, on the other hand, is used to build and test projects, which is closer to the task of identifying reproducible bugs.

## 3.5 Final Remarks

In this chapter, we presented the BEARS project, including its approach to collect and store bugs (the BEARS-COLLECTOR) and a collection of 251 bugs from 72 projects (the first version of the BEARS-BENCHMARK). The BEARS-COLLECTOR performs attempts to reproduce buggy and patched program versions from builds in Travis CI: builds can be scanned from specific projects given a time window or from any project in real time.

We designed BEARS to maximize the automation for collecting and storing bugs from a diversity of projects. We have also made it extensible for the research community to contribute with new bugs to be added in the BEARS-BENCHMARK. Our intention is to have a community-driven benchmark, where the community spends time and effort to minimize all kinds of bias. Nonetheless, the adoption and contribution of an open initiative is a complex phenomenon that we cannot yet evaluate.

Chapter 4

# A Descriptive Study on Bug Benchmarks

Building benchmarks of bugs is a challenging task. Despite the efforts of the authors of bug benchmarks, such benchmarks generally do not include detailed information about the projects the bugs came from, the bugs themselves, and their patches, so advanced evaluation of repair tools becomes more difficult. We highlight three tasks that could make the evaluation of repair tools more robust:

- *Selection of bugs* is used to filter out bugs that do not belong to the bug class that a repair tool under evaluation targets. For instance, NPEFix (DURIEUX et al., 2017) is a tool specialized in null pointer exception fixes and will probably fail on bugs not fixed by a human with null pointer checking. So, a coherent and fair evaluation would include only bugs within the target bug classes of the tools under evaluation.

- *Correlation analysis* is an advanced analysis of the results produced by a repair tool, making it possible to derive conclusions such as "the repair tool performs well on bugs that have the property X". This kind of analysis requires a characterization of all bugs available in the dataset used.

- *Explanatory analysis* is also an advanced analysis of the results produced by a repair tool, making it possible to explain and discuss the results. This kind of analysis is mainly useful when there is no selection of bugs before the evaluation.

To support these tasks, we present in this chapter two main contributions towards the characterization of benchmarks of bugs. First, we present ADD (Automatic Diff Dissection), a tool that extracts, given a patch, four types of properties defined in a previous taxonomy (Section 4.1). Then, we present a characterization of Defects4J, Bugs.jar, and Bears, where we extracted information on the projects from which the bugs came from, the bugs themselves, as well as the human-written patches using ADD (Section 4.2).

## 4.1 ADD: Supporting the Characterization of Patches

In a previous work (SOBREIRA et al., 2018), we *manually* analyzed the *patches* of the Defects4J benchmark (JUST et al., 2014). As a result, we delivered a taxonomy of properties and the annotation of the Defects4J patches according to the taxonomy. Despite the value of our previous manual work, analyzing patches from different benchmarks, such as BEARS and Bugs.jar (SAHA et al., 2018), is tedious and time-consuming. Nevertheless, the taxonomy already built is a useful resource to guide the automation of patch analysis.

### 4.1.1 The Taxonomy of Properties related to Patches

The taxonomy we defined in our previous work (SOBREIRA et al., 2018) includes, broadly, four types of properties regarding patches:

*Patch size.* The metric *patch size* is the sum of three other metrics: *the number of added lines*, *the number of deleted lines*, and *the number of modified lines* (consecutive pairs of added and deleted lines). Listing 1 shows an example of a patch with one modified line (line 635), two non-paired removed lines (the old 636 and 639 lines), and no non-paired added line. By summing up these lines, we have the metric *patch size*, which in the example is 3 lines.

*Patch spreading.* The spreading of a patch in the source code can be measured using five metrics. First, there is *the number of chunks*: a chunk is a sequence of continuous changes in a file, consisting of the combination of the addition, removal, and modification of lines. Listing 1 has two chunks: the first one is composed of lines 635 and 636 and the second one is composed of the old line 639. The second metric is *spreading of chunks* in a patch, which is the number of lines interleaving chunks in the patch. In Listing 1, there are only two lines between the old line 636 (end of the first chunk) and the old line 639 (beginning of the second chunk), which is the value of the metric spreading of chunks in this case. The remaining three metrics for patch spreading are *the number of modified files*, *classes*, and *methods*.

*Repair actions.* There are several types of actions that are performed on source code elements to produce a patch. *Repair actions* are the basic building blocks for patches. For example, the patch in Listing 1 was obtained through the actions *modification of method call parameter value* (line 635) and *removal of conditional (`if`) branch* (old lines 636 and 639). Repair actions provide fine-grained information beyond the simple counting of addition, removal, and modification of lines.

*Repair patterns.* Abstractions that occur recurrently in patches, which can involve combinations of repair actions, are called *repair patterns*. For example, the modified line 635 in Listing 1 illustrates the repair pattern *Constant Change*, while the removed lines 636 and 639 illustrate the repair pattern *Unwraps-from* `if`.

```
635      - JsName name = getName(ns.name, false);
     635 + JsName name = getName(ns.name, true);
636      - if (name != null) {
637  636    refNodes.add(new ClassDefiningFunctionNode(
638  637        name, n, parent, parent.getParent()));
639      - }
```

Listing 1 – Human-written patch for bug Closure-40 from Defects4J.

Observe that our description of the taxonomy is based on four types of properties: patch size, patch spreading, repair actions, and repair patterns. However, this description is just an overview of the taxonomy and its full description can be found in our previous work (SOBREIRA et al., 2018). Table 7 summarizes the number of actual properties for each property type. For the property types related to metrics (patch size and spreading), Table 7a shows the actual number of metrics in each of the two property types. For instance, *patch size* includes four metrics, which are the number of added lines, the number of deleted lines, the number of modified lines, and the patch size itself. For repair actions and patterns, Table 7b shows the number of existing groups and the total number of actual actions and patterns contained in these groups. For instance, there are nine groups of repair patterns, totaling 25 patterns. One group of patterns is *Conditional Block*, which contains the four patterns *Conditional block addition*, *Conditional block addition with return statement*, *Conditional block addition with exception throwing*, and *Conditional block removal*.

Table 7 – Patch property types and their sizes.

(a) Patch size and spreading.

| Property | # Metrics |
|---|---|
| Patch size | 4 |
| Patch spreading | 6 |

(b) Repair actions and patterns.

| Property | # Group | # Variants |
|---|---|---|
| Repair actions | 10 | 50 |
| Repair patterns | 9 | 25 |

### 4.1.2 Implementation

To support the characterization of patches from benchmarks of bugs based on the four patch properties, we developed the ADD tool. Its purpose falls in the source code change analysis task. ADD uses a combination of source code change analysis at the file, line, and Abstract-Syntax Tree (AST) levels, depending on the patch property to be calculated or detected. Figure 9 shows an overview of ADD. Given as input the buggy program source code and the patch file (i.e., the diff file), ADD calculates the metrics on patch size and spreading and detects repair actions and patterns, which are all packaged in a JSON file containing the results on the properties. ADD is separated into two components: the metric calculator and the detector. We explain these two components in the next sections.

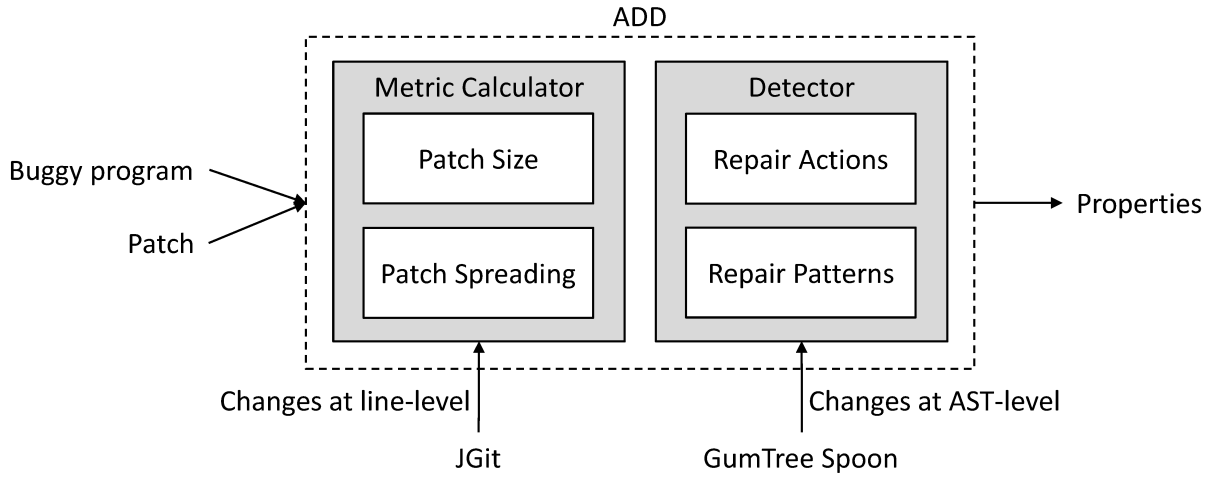


Figure 9 – Overview of the ADD tool.

#### 4.1.2.1 The Metric Calculator

*The metric calculator* is responsible for calculating the patch size and spreading metrics and performs source code change analysis mostly at the line level, as well as at the file level to calculate the number of modified files. The metric calculator uses JGit[9], an implementation of the Git version control system in Java, to retrieve information on files and source code lines changed by a patch. For instance, to calculate patch size, the metric calculator analyzes each block of changes (sequential changed lines) in the buggy program version to calculate the number of deleted lines and in the patched program version to calculate the number of added lines. Since the changes are sequential, the number of deleted lines minus the number of added lines is the number of modified lines. By summing up all added, deleted, and modified lines, the patch size is calculated.

#### 4.1.2.2 The Detector

*The detector* is responsible for detecting instances of repair actions and patterns and performs source code change analysis at the AST level. It consists of two main tasks for a given patch: the retrieval of the AST diff and its analysis. The former is the same process for the detection of both repair actions and patterns, but the latter is different for the two types of properties.

The *retrieval of the AST diff* works as follows. Given as input the buggy version of the program and the patch file, the detector retrieves the AST diff between the buggy and patched code (also known as *edit scripts*) using the GumTree algorithm (FALLERI et al., 2014). There are different implementations of the GumTree algorithm, and we use GumTree Spoon[10] since this tool provides the AST diff nodes in the representation of the Spoon library (PAWLAK et al., 2016), which is based on a well-designed meta-model to

[9] JGit: <https://github.com/eclipse/jgit>. Latest access: December 13, 2018.
[10] GumTree Spoon AST Diff: <https://github.com/SpoonLabs/gumtree-spoon-ast-diff>. Latest access: December 13, 2018.

represent Java programs. Therefore, we can analyze the edit scripts returned by GumTree with Spoon.

The *analysis of the retrieved AST diff* to detect *repair actions* is performed with the visitor- and scanner-based query API provided by Spoon. This means that each AST node in the AST diff is visited and we can detect the different repair actions according to the different types of AST node. We choose one repair action, *Method call addition*, to use as an example to describe in detail how the detector performs the automatic detection of a repair action. Consider the source code diff in Listing 2. In the fixed version of this code, a method call was added in line 65, `ImageMapUtilities.htmlEscape`, wrapping the variable `toolTipText`. The added method call is an instance of the repair action *Method call addition*. To detect this repair action, the detector overrides the visit method `visitCtInvocation` from the Spoon library, which is used to visit AST nodes of the type `CtInvocation`. If the AST node `CtInvocation` is an added node in the AST diff, the detector has found an instance of the *Method call addition* repair action.

```
64 64   public String generateToolTipFragment(String toolTipText) {
65    -     return " title=\"" + toolTipText
   65 +     return " title=\"" + ImageMapUtilities.htmlEscape(toolTipText)
66 66           + "\" alt=\"\"";
67 67 }
```

Listing 2 – Human-written patch for bug Chart-10 from Defects4J.

The *analysis of the retrieved AST diff* to detect *repair patterns* is more complex than the analysis to detect repair actions since they are not fine-grained structures. The detector contains a set of sub-detectors, one for each repair pattern group, because each group has its own definition, which requires specific detection strategies[11]. The strategies are mainly based on searching and checking code elements or structures in the AST diff. We choose one pattern, *Missing Null-Check*, to be used as an example to describe how the detector performs automatic detection in detail. Given the edit scripts from a patch, the strategy of the *Missing Null-Check* detector is the following:

1. It searches for the addition of a binary operator where one of the two elements is `null`, i.e., a null-check;

2. It extracts from the null-check the variable being checked (`variable <operator> null`) or the variable being used to call a method where its return is being checked (`variable.methodCall() <operator> null`);

3. It verifies if the extracted variable is new, i.e., was added in the patch. If the variable is not new, a missing null check was found. Otherwise, it verifies if the new null check wraps existing code. If it does, a missing null check was found.

[11] The detector was designed in a modularized way that makes it possible to easily implement a strategy for the detection of a new pattern by extending an existing class.

Consider the source code diff in Listing 3. In the buggy version of the code, in the old line 2166, a null pointer exception had been thrown when the variable `markers` was null and accessed for a method call. In the fixed version, a conditional block was added to check whether `markers` is null, and in such case, the method returns, so the program execution does not reach the point of the exception. The added null check is an instance of the pattern *Missing Null-Check*. Note that the null check was added in a new conditional block, so this patch also contains an instance of the pattern *Conditional Block Addition with Return Statement*. Additionally, this conditional block was added at four different locations in the code (see Chart-14), which consists in the *Copy/Paste* pattern.

```
2166      + if (markers == null) {
2167      +     return false;
2168      + }
2166 2169   boolean removed = markers.remove(marker);
```

Listing 3 – Human-written patch for bug Chart-14 from Defects4J.

A missing null check can appear in different ways beyond the addition of an entire conditional block. In Listing 4, for instance, the missing null check was added by wrapping an existing block of code. This type of change consists of the pattern *Wraps-with if*.

```
1191 1191   ChartRenderingInfo owner = plotState.getOwner();
     1192 + if (owner != null) {
1192 1193      EntityCollection entities = owner.getEntityCollection();
1193 1194      if (entities != null) {
1194 1195         entities.add(new AxisLabelEntity(this, hotspot,
1195 1196                 this.labelToolTip, this.labelURL));
1196 1197      }
     1198 + }
```

Listing 4 – Human-written patch for bug Chart-26 from Defects4J.

Since our detector searches for binary operators involving null checks, it also detects missing null checks in other structures beyond `if` conditionals. In Listing 5, for instance, there is an example of a conditional structure using the ternary operator. When the ternary operator is used, and an existing expression is placed in the `then` or `else` expression, we have the pattern *Wraps-with if-else*. Note that this patch is also an instance of the pattern *Single Line* since only one line was affected by the patch.

```
29    - description.appendText(wanted.toString());
   29 + description.appendText(wanted == null ? "null" : wanted.toString());
```

Listing 5 – Human-written patch for bug Mockito-29 from Defects4J.

For these three patches used as examples, ADD was able to detect all existing instances of the mentioned patterns above.

### 4.1.3 Evaluation Design

Our evaluation consists of running ADD on real patches to measure its ability to calculate metrics and detect instances of the 50 repair actions and the 25 repair patterns.

*Subject dataset.* The patches used as input to ADD are from Defects4J (JUST et al., 2014), which consists of 395 patches from six real-world projects. We chose this benchmark since all of its patches have been annotated with the four properties implemented in ADD in our previous work (SOBREIRA et al., 2018), where the metrics were calculated using scripts and the repair actions and patterns were manually detected. By choosing this benchmark, we can directly compare the results generated by ADD with the results from previous work.

*Result analysis.* After running ADD on the Defects4J patches, we analyzed the results as follows. For patch size and spreading, we calculated the *accuracy* of ADD using our previous results as an oracle. For repair actions and patterns, we calculated the *precision* and *recall* of ADD, also using the previous manual detection as an oracle. For the repair actions, we performed a direct calculation. For the repair patterns, we performed an advanced analysis of the results to find the ground truth, i.e., if ADD actually missed or wrongly detected patterns. We performed this advanced analysis because repair patterns are harder to detect both manually and automatically, so mistakes in the previous manual analysis could have been made. For the sake of keeping this chapter as clear as possible, we report on that advanced analysis in Appendix A as additional content while keeping this thesis complete.

### 4.1.4 Evaluation Results and Discussion

The overall results of the evaluation of ADD are presented in Table 8. The number for each property type is an average of the results of all sub-properties. For instance, patch size includes four metrics on patch size, i.e., added lines, deleted lines, modified lines, and finally patch size. The accuracy we show in Table 8 for patch size is on all these four sub-properties.

Table 8 – Overall performance of ADD.

(a) Metric calculator.

| Property | Accuracy |
|---|---|
| Patch size | 99% |
| Patch spreading | 97% |

(b) Detector.

| Property | Precision | Recall |
|---|---|---|
| Repair actions | 78% | 82% |
| Repair patterns | 91% | 92% |

We observe that the accuracy values for patch size and spreading are almost 100%. This is because these metrics are straightforward: their calculation does not even need analysis at the AST level. Moreover, the previous work used as an oracle calculated

most of the metrics on patch size and spreading using scripts, so it was already somehow automated, which may also explain why we have such a high accuracy in ADD.

In terms of the detection of repair actions and patterns, the numbers are not that close to 100%. This was expected because the detection of specific changes in the source code is a more difficult task than the calculation of metrics. We observed several reasons for missing the detection of repair action and pattern instances and also for wrongly detecting non-exiting instances. Appendix A contains a detailed report on the advanced analysis we performed on the detection of repair patterns as additional material. However, we report here on a specific problem we faced and that is still an open challenge.

The GumTree algorithm is a robust algorithm that finds mappings between two ASTs and returns an AST diff. The AST nodes in the diff correspond to one of the four operations: insertion, deletion, updating, or moving. The mapping of AST nodes is performed only between AST nodes of the same type. For instance, for the patch in Listing 6, the AST nodes retrieved by GumTree are *updating* nodes, i.e., there are known mappings between the variables `paint` to `outlinePaint` and between the variables `stroke` to `outlineStroke`. The detector was able to detect the repair action *Variable replacement by another variable* in such a patch.

```
95     - super(paint, stroke, paint, stroke, alpha);
   95  + super(paint, stroke, outlinePaint, outlineStroke, alpha);
```

Listing 6 – Human-written patch for bug Chart-20 from Defects4J.

Another case about the repair action *Variable replacement by another variable* is shown in Listing 7. It is possible to see that the variable `RegularTimePeriod.DEFAULT_TIME_ZONE` was replaced by `zone`. However, the detector did not succeed in finding the instance of that repair action in this patch. This happened because GumTree was unable to find an *updating* mapping between the two variables. Instead, it returns a *deletion* operation and an *insertion* one. The mapping is not found because `RegularTimePeriod.DEFAULT_TIME_ZONE` is a class variable while `zone` is a local variable. We could not handle this case yet. A workaround could be to try and find the mapping of the two operations using the lines on which they were applied in the source code. This would work for the patch in Listing 7. However, this is a naive approach that would not work in most cases because other changes performed in the source code file can affect the line numbers of posterior source code.

```
175     - this(time, RegularTimePeriod.DEFAULT_TIME_ZONE, Locale.getDefault());
    175 + this(time, zone, Locale.getDefault());
```

Listing 7 – Human-written patch for bug Chart-8 from Defects4J.

## 4.2 Characterizing Benchmarks

The second part of this chapter is dedicated to presenting a descriptive study on benchmarks of bugs. Recall from the motivation of this study, presented at the beginning of this chapter, that three tasks could make the evaluation of repair tools more robust: *selection of bugs*, *correlation analysis*, and *explanatory analysis*. This descriptive study, which results in information about the bugs in benchmarks, supports these tasks. Moreover, this study results in knowledge that can be useful for helping researchers measure the representativeness of the benchmarks and find their flaws, learn about real bugs and, by extension, improve their repair approaches and tools.

### 4.2.1 Research Questions

Our study on benchmarks is guided by the following three research questions:

**RQ #1**. [Projects] What are the types of the projects from which the bugs come?

Characterizing the projects is important because benchmarks of bugs should contain bugs from a diversity of projects (LE GOUES et al., 2013). The goal of this research question is to categorize bugs according to their project types, which can be further used especially for *correlation analysis* on repair tool evaluations. We classify projects into library/application and single/multi-module.

**RQ #2**. [Tests] How are the bugs exposed?

Test-suite-based repair tools use the test suite of the program under repair to repair a bug. Knowledge of how the bugs of the benchmarks are exposed by test cases can be useful for all the tasks of *selection of bugs*, *correlation analysis*, and *explanatory analysis*. We collect and analyze the exception types thrown by the failing test cases, e.g., `AssertionError` and `NullPointerException`.

**RQ #3**. [Patches] How are the patches written by developers to fix the bugs?

The majority of repair tools try to repair a bug in a limited way (e.g., only in one source code location) or with a specific target (e.g., Nopol fixes `if` conditions). These tools might fail to repair bugs that were repaired by developers in a different way than how the repair tools try to fix bugs. Therefore, information on how the bugs were fixed by developers can help in the *selection of bugs* for repair tool evaluations, so that the computational power required is reduced, and in *explanatory analyses* on the results of repair tool evaluations. We analyze the human-written patches contained in the benchmarks with ADD, which calculates patch size and spreading and identifies instances of repair actions and patterns.

### 4.2.2 Subject Benchmarks of Bugs

There are several benchmarks of bugs in the literature, as presented in Section 2.2, and the one introduced in this thesis, BEARS. To select benchmarks of bugs for our study, we defined the following inclusion criteria:

*Criterion #1.* The benchmark must contain bugs in the Java language. This criterion excludes ManyBugs (LE GOUES et al., 2015), IntroClass (LE GOUES et al., 2015), and Codeflaws (TAN et al., 2017).

*Criterion #2.* The benchmark must contain bugs that were originally found in Java code, as opposed to code in other languages that were transpiled to Java. This criterion excludes IntroClassJava (DURIEUX; MONPERRUS, 2016b) and QuixBugs (LIN et al., 2017).

*Criterion #3.* The benchmark must be peer-reviewed, presented in a research paper dedicated to it. This criterion excludes PARDataset (KIM et al., 2013), ConditionDataset (XUAN et al., 2016), and NPEDataset (DURIEUX et al., 2017).

*Criterion #4.* The benchmark must include, for each bug, at least one failing test case. This criterion excludes iBugs (DALLMEIER; ZIMMERMANN, 2007).

After searching the literature for benchmarks that meet our criteria, we ended up with three benchmarks. Table 9 presents the selected benchmarks, which are BEARS, Bugs.jar, and Defects4J.

Table 9 – Selected benchmarks of bugs.

| Benchmark | # Projects | # Bugs | Commit mining strategy |
|---|---|---|---|
| BEARS (from this thesis) | 72 | 251 | with continuous integration |
| Bugs.jar (SAHA et al., 2018) | 8 | 1,158 | with bug tracker |
| Defects4J (JUST et al., 2014) | 6 | 395 | with bug tracker |

### 4.2.3 Data Analysis and Results

In this section, each research question is studied in a dedicated section, which is separated into two parts: data collection and results.

#### 4.2.3.1 Analysis on Projects (RQ #1)

**Data collection.** Projects can be categorized in different aspects. Our categorization includes two of them. First, we classified the projects by *domain* considering the categories *library* and *application*, which are considered in previous work (RAY et al., 2017). Second, we classified projects by *project structure* considering the number of modules that a project

Table 10 – The number of projects divided into libraries and applications and the proportion of multi-module projects in the benchmarks.

| | # Projects | # Library | # Application | % Multi-module projects |
|---|---|---|---|---|
| Bears | 72 | 49 (68%) | 23 (32%) | 56.9% |
| Bugs.jar | 8 | 6 (75%) | 2 (25%) | 87.5% |
| Defects4J | 6 | 5 (83%) | 1 (17%) | 0% |

Table 11 – The number of bugs from the projects divided into libraries and applications and the proportion of bugs from multi-module projects in the benchmarks.

| | # Bugs | # Library | # Application | % Multi-module projects |
|---|---|---|---|---|
| Bears | 251 | 167 (67%) | 84 (33%) | 26.69% |
| Bugs.jar | 1,158 | 1,012 (87%) | 146 (13%) | 87.31% |
| Defects4J | 395 | 262 (66%) | 133 (34%) | 0% |

has, i.e., if it is a single or a multi-module project. To classify a project as a library or an application, we performed a manual analysis on the `README.md` file contained in its GitHub repository. To classify a project as single or multi-module, we used a plugin[12] that returns information on Maven projects, such as the project module names.

**Results.** Table 10 and Table 11 present the results of our data collection. The first table contains the results in number of projects and the second table contains the same information but at the bug level. At the project level, we observe that the division of projects in libraries and applications is more balanced in Bears. However, considering such a division at the bug level, Defects4J is more balanced. The majority of bugs in Bugs.jar are from library projects (87%), which means that a low rate of bugs comes from applications. On multi-module projects, we observe that Defects4J is the only benchmark that does not have any. The majority of projects in Bears and Bugs.jar are multi-module. However, at the bug level, we observe that Bears contains a low proportion of bugs from multi-module projects (∼27%).

**Answer to RQ #1**. **What are the types of the projects from which the bugs come?** All benchmarks contain bugs from libraries and applications, although most bugs come from libraries. Both Bears and Bugs.jar contain bugs from multi-module projects. However, Defects4J does not have any. Multi-module projects are more complex than single-module ones, to both build and test, because modules depend on each other. Because this architecture has been widely adopted in open-source projects, benchmarks should include bugs from multi-module projects for the evaluation of program repair tools, and repair tool developers should include support for multi-module projects in their tools.

[12] Project Info Maven Plugin: <https://github.com/tdurieux/project-info-maven-plugin>. Latest access: December 13, 2018.

#### 4.2.3.2 Analysis on Tests (RQ #2)

**Data collection.** To answer RQ #2, we analyzed the test cases that expose the bugs of the benchmarks. First, we executed the complete test suite of the buggy programs under investigation. At the end of the executions, we collected the number of failing test cases and the exceptions that were triggered during their executions. There are two types of exceptions: *test failure* and *test in error*. A test failure is reported when an asserted condition in a test case does not hold (LANGR et al., 2015), i.e., when an expected result value does not match the actual value. On the other hand, a test in error is reported when an exception is thrown and is not caught during the execution of the test (LANGR et al., 2015), e.g., an array index out of bounds.

**Results.** Table 12 shows the distribution of the number of failing test cases per benchmark. We observed that the majority of bugs of all benchmarks are reproduced with a single failing test case, and this number increases to 2 only in the third quartile. However, the maximum number of failing test cases for a single bug is 145, 71, and 66, respectively, for BEARS, Bugs.jar, and Defects4J.

Table 12 – Distribution of the number of failing test cases per benchmark.

| | Min | 25% | 50% | 75% | Max | Avg |
|---|---|---|---|---|---|---|
| BEARS | 1 | 1 | 1 | 2 | 145 | 3 |
| Bugs.jar | 0 | 1 | 1 | 2 | 71 | 2 |
| Defects4J | 1 | 1 | 1 | 2 | 66 | 2 |

Table 13 presents the number of bugs that were reproduced with test failures, tests in error, and both test failures and tests in error. We were unable to reproduce some bugs, so the sum of the columns does not result in the total number of bugs of the benchmarks. We observed that the majority of bugs in the three benchmarks are reproduced with test failures, i.e., by assertions that specify the correct behavior of the program, such as "*x should be equal to 0*", not being held. This is valuable information for studies in test-suite-based automatic program repair because different types of bug exposure might require different repair strategies. A test in error requires handling uncaught exceptions, while a test failure requires that an asserted condition holds.

Table 13 – The number of bugs reproduced with test failures and tests in error.

| | # Test failure | # Test in error | # Both |
|---|---|---|---|
| BEARS | 137 | 80 | 31 |
| Bugs.jar | 619 | 185 | 117 |
| Defects4J | 278 | 98 | 18 |

Table 14 presents the ranking of the most frequent exceptions that were thrown when the bugs of the benchmarks were reproduced. We observed that `AssertionError` is, by far, the most frequent exception in the three benchmarks, followed by `ComparisonFailure`. Both of them are *test failures*. On *test in error*, the most frequent exception is `NullPointerException`, which is one of the most frequent exceptions in production environments. Finally, the table also shows the diversity of exception types that are thrown when exposing the bugs of the benchmarks. We detected 47, 82, and 35 different exceptions in Bears, Bugs.jar, and Defects4J, respectively.

Table 14 – The most frequent exceptions triggered by failing test cases per benchmark.

| # | Bears | | Bugs.jar | | Defects4J | |
|---|---|---|---|---|---|---|
| 1 | AssertionError | (133) | AssertionError | (458) | AssertionFailedError | (258) |
| 2 | ComparisonFailure | (30) | ComparisonFailure | (229) | ComparisonFailure | (47) |
| 3 | NullPointerException | (29) | AssertionFailedError | (97) | NullPointerException | (17) |
| 4 | AssertionFailedError | (13) | NullPointerException | (51) | IllegalArgumentException | (15) |
| 5 | SpoonException | (9) | WicketRuntimeException | (32) | ArrayIndexOutOfBoundsException | (10) |
| 6 | NumberFormatException | (8) | IllegalStateException | (25) | StringIndexOutOfBoundsException | (7) |
| 7 | JsonMappingException | (8) | NoClassDefFoundError | (20) | IllegalStateException | (7) |
| 8 | IllegalArgumentException | (6) | Exception | (19) | ClassCastException | (7) |
| 9 | IndexOutOfBoundsException | (5) | IllegalArgumentException | (16) | RuntimeException | (6) |
| 10 | IllegalStateException | (5) | CouldNotLockPageException | (15) | IllegalFieldValueException | (5) |
| | 37 more | | 72 more | | 25 more | |

**Answer to RQ #2**. **How are the bugs exposed?** Test-suite-based automatic program repair approaches rely on the failing test cases of a program under repair to fix a bug. However, we observed that most of the reproduced bugs contain only one failing test case. Having only one failing test is potentially not enough to fully specify the incorrect behavior of a program. Moreover, according to the number of test cases in error and the ranking of exception types, specialized approaches targeting the most frequent exceptions might be beneficial. To our knowledge, only NPEFix (Durieux et al., 2017) targets a specific exception type, i.e., `NullPointerException`.

#### 4.2.3.3 Analysis on Patches (RQ #3)

**Data collection.** To extract information from the patches written by developers to fix the bugs of the benchmarks, we used ADD, which is presented in the first part of this chapter. Recall that ADD calculates metrics on patch size and spreading and detects instances of repair actions and patterns in patches. We executed ADD on all bugs from the three benchmarks, and the gathered results are from this execution. Note that we already had the Defects4J patches annotated with all patch properties from previous work (Sobreira et al., 2018). However, we did not use such data. We preferred to run ADD on Defects4J because ADD is not 100% precise, and comparing the manually obtained data on Defects4J with automated obtained data on other benchmarks (i.e., Bears and Bugs.jar) could result in a biased comparison.

Table 15 – Descriptive statistics on the size and spreading of the patches included in the benchmarks.

| | Bears | | | | | Bugs.jar | | | | | Defects4J | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Min | 25% | 50% | 75% | Max | Min | 25% | 50% | 75% | Max | Min | 25% | 50% | 75% | Max |
| # Added lines | 0 | 1 | 4 | 11 | 244 | 0 | 1 | 6 | 16 | 831 | 0 | 0 | 2 | 6 | 48 |
| # Removed lines | 0 | 0 | 0 | 1 | 142 | 0 | 0 | 0 | 3 | 389 | 0 | 0 | 0 | 0 | 24 |
| # Modified lines | 0 | 1 | 2 | 4 | 32 | 0 | 1 | 2 | 6 | 196 | 0 | 0 | 1 | 2 | 27 |
| Patch size | 1 | 3 | 8 | 18 | 312 | 1 | 4 | 11 | 26 | 936 | 1 | 2 | 4 | 9 | 54 |
| # Chunks | 1 | 1 | 2 | 5 | 23 | 1 | 2 | 3 | 7 | 141 | 1 | 1 | 2 | 3 | 20 |
| Spreading | 0 | 0 | 8 | 75 | 1,548 | 0 | 0 | 21 | 114 | 3,673 | 0 | 0 | 1 | 18 | 1,332 |
| # Files | 1 | 1 | 1 | 1 | 10 | 1 | 1 | 1 | 1 | 13 | 1 | 1 | 1 | 1 | 7 |
| # Classes | 0 | 1 | 1 | 1 | 7 | 0 | 1 | 1 | 2 | 14 | 1 | 1 | 1 | 1 | 6 |
| # Methods | 0 | 1 | 1 | 2 | 12 | 0 | 1 | 1 | 3 | 50 | 0 | 1 | 1 | 2 | 19 |

**Results on patch size and spreading.** Table 15 presents the distribution of patch size and patch spreading for Bears, Bugs.jar, and Defects4J. These metrics provide an overview of the size of the patches written by the developers.

Findings: We observed that patches from Defects4J are smaller than patches from Bears and Bugs.jar. In fact, around 75% of the Defects4J patches are smaller than nine lines of code, which is twice less than the Bears patches and three times less than the Bugs.jar patches. We also observed that the developers more frequently created patches to fix the bugs by adding new lines than by removing and modifying existing ones. In terms of spreading-related metrics, the Defects4J patches are also more localized than the Bears and Bugs.jar patches. There might be several explanations for this. One of them is that the Defects4J bugs might require simpler quick fixes. Another explanation may be related to the Defects4J creation strategy. The Defects4J patches were manually isolated from changes unrelated to the bug fixes (e.g., refactorings) and, consequently, the size of the patches was reduced.

Implications: The majority of program repair approaches perform single-point repair and focus their strategies on modifying or suppressing existing lines of code. This is a limitation of the current repair approaches, as the patches written by developers do not share these characteristics to a large extent. Our results show that there is a need for multi-point program repair, such as the ones performed by NPEFix (Durieux et al., 2017) and ARJA (Yuan; Banzhaf, 2018).

**Results on repair actions and patterns.** Repair actions and patterns give an overview of how developers created the patches to fix the bugs. Table 16 presents the number of detected repair patterns and Table 17 presents the number of detected repair actions for each benchmark.

Findings: We observed that the patches in the three benchmarks follow a similar distribution of repair patterns by analyzing Table 16. *Conditional Block* and *Wraps-with/Unwraps-*

Table 16 – The occurrence of repair pattern groups per benchmark.

| Pattern group | BEARS | Bugs.jar | Defect4J |
|---|---|---|---|
| Conditional Block | 48% (121) | 50% (586) | 47% (187) |
| Wraps-with/Unwraps-from | 34% (85) | 34% (395) | 31% (123) |
| Wrong Reference | 30% (77) | 32% (376) | 20% (79) |
| Expression Fix | 23% (59) | 26% (306) | 29% (114) |
| Copy/Paste | 17% (43) | 27% (317) | 18% (72) |
| Missing Null-Check | 20% (50) | 15% (176) | 12% (48) |
| Single Line | 16% (42) | 13% (152) | 24% (96) |
| Constant Change | 15% (38) | 13% (159) | 5% (21) |
| Code Moving | 7% (18) | 8% (100) | 2% (10) |

*from* patterns are the most frequently found in the patches of all the benchmarks. However, the occurrence rate of the patterns *Single Line* and *Wrong Reference* are considerably different in Defects4J. *Single Line* is more present in Defects4J than in the other benchmarks (which aligns with the findings related to patch size and spreading presented above), and *Wrong Reference* is less frequent in Defects4J.

Regarding repair actions, we observed that the top-3 most performed repair actions to fix the bugs is composed of the same repair actions for the three benchmarks. These repair actions are the addition of method call, conditional `if`, and assignment. In terms of repair action groups, we observed that repair actions related to *Method Call* are the most frequent ones in the top-10 of all benchmarks.

Table 17 – The top-10 most performed repair actions in patches per benchmark. "Add" is an abbreviation of Addition.

| # | BEARS | | Bugs.jar | | Defects4J | |
|---|---|---|---|---|---|---|
| 1 | Method Call Add | (69%) | Method Call Add | (68%) | Method Call Add | (59%) |
| 2 | Conditional `if` Add | (42%) | Assignment Add | (46%) | Conditional `if` Add | (42%) |
| 3 | Assignment Add | (38%) | Conditional `if` Add | (41%) | Assignment Add | (34%) |
| 4 | Variable Add | (31%) | Variable Add | (38%) | Variable Add | (25%) |
| 5 | Return Add | (30%) | Method Call Removal | (30%) | Return Add | (23%) |
| 6 | Method Call Removal | (28%) | Return Add | (29%) | Conditional `if-else` Add | (20%) |
| 7 | Method Call Replacement | (26%) | Method Call Replacement | (27%) | Method Call Removal | (18%) |
| 8 | Conditional `if-else` Add | (21%) | Conditional `if-else` Add | (23%) | Method Call Replacement | (15%) |
| 9 | Method Call Parameter Add | (20%) | Object Instantiation Add | (22%) | Object Instantiation Add | (15%) |
| 10 | Object Instantiation Add | (20%) | Conditional Removal | (21%) | Conditional Removal | (14%) |

Implications: Several repair tools focused mainly on conditional statements, e.g., the tools Nopol (XUAN et al., 2016), DynaMoth (DURIEUX; MONPERRUS, 2016a), and ACS (XIONG et al., 2017). However, handling method calls with rich side-effects has been rather neglected. To our knowledge, only generate-and-validate approaches (MARTINEZ; MONPERRUS, 2016; YUAN; BANZHAF, 2018) and DynaMoth (DURIEUX; MONPERRUS, 2016a) are capable of synthesizing such patches. Our findings suggest that research on repair strategies to handle method calls should be conducted since repair actions on method calls are frequently found in patches that fixed the bugs of the three benchmarks.

On the repair patterns, not all of them are at the same level of abstraction, which means that creating repair tools to synthesize patches with certain patterns might be difficult due to their generic nature. However, we have identified repair patterns that are precise enough to be targeted by implementations of repair tools. For instance, a variant of the repair pattern *Conditional Block* is *Conditional (if) Block Addition with Exception Throwing*. Patches with this pattern can be synthesized, as confirmed by recent work on ACS (XIONG et al., 2017).

**Answer to RQ #3**. **How are the patches written by developers to fix the bugs?** The patches written by developers, which are used in automatic repair research as ground truth to verify the correctness of generated patches, contain mostly the addition of code in the three benchmarks. Moreover, the patches for the bugs in Defects4J are smaller and less spread than the patches for the bugs in BEARS and Bugs.jar. Approximately half of the patches in the three benchmarks contain instances of the repair patterns that are related to *Conditional Block*. The most frequent repair action group, also in the three benchmarks, is related to method calls. These findings might help researchers make decisions for defining new strategies and implementing new repair tools to be able to synthesize patches with similar characteristics to human-written patches.

## 4.3 Discussion

### 4.3.1 Threats to Validity

We evaluated ADD on patches from Defects4J. However, since Defects4J may not be representative of all different cases of fixing bugs in the world, it is possible that ADD still cannot generalize for systems including patches that differ a lot from those in Defects4J.

The ultimate precision and recall calculated when evaluating the performance of the ADD detector are based on a manual disagreement analysis (see Appendix A). This analysis can be subject to small errors and misconceptions, typical of any manual work. To mitigate this, such analysis was conducted, for each repair pattern group, by the author of this thesis and two collaborators, in live discussion sessions.

Although this study provides an overview of three benchmarks of bugs, the composition of these benchmarks may not be representative enough to allow broad generalization. To investigate whether the findings can be generalized to all bugs, a follow-up of this work is required.

### 4.3.2 Related Work

**Patch analysis of bug benchmarks.** Motwani et al. (2018) performed a characterization of the Defects4J bugs. Similarly to our work, they annotated Defects4J bugs with

patch size and number of modified files. On the characteristics of the patches, they annotated the bugs with nine code modification types, such as whether the patch contains the addition of method calls, which are similar to our repair actions. However, our taxonomy of repair actions is more comprehensive and fine-grained, since we arranged the actions in groups considering more detailed changes. For instance, instead of having the information that a patch changed arguments in a method call, we have the information that an argument was added or removed, or that an argument value was changed or swapped with another one in a method call. Moreover, Motwani et al. considered other information than us, such as the number of relevant test cases, which makes our work and their work complementary to each other.

**Patch analysis at AST level.** Martinez et al. (2013) also reported on automatic detection of bug fix patterns at the AST level. The main differences between their work and our work are the following. First, they focused on 18 bug fix patterns from Pan et al. (2009) while we focused on the 25 patterns from our previous work (SOBREIRA et al., 2018). Second, they used the ChangeDistiller AST differencing algorithm (FLURI et al., 2007) while we used GumTree (FALLERI et al., 2014). The latter outperforms the former by maximizing the number of AST node mappings, minimizing the edit script size, and detecting better move actions (FALLERI et al., 2014). Moreover, they pointed out that ChangeDistiller works at the statement level, preventing the detection of certain fine-grained patterns. Third, they formalized a representation for change patterns and used this representation to specify patterns. Then, to detect a pattern, a match of its specification must be found in a given edit script. However, such representation is based on change type (e.g., addition) over code elements (e.g., `if`), which does not support the specification of patterns such as *Single Line*.

There is also another work that detects repair actions at the AST level, presented by Liu et al. (2018a). Their study was on a large scale, involving 16,450 bug-fixing commits from seven Java open-source projects. Like us, they also used GumTree (FALLERI et al., 2014). Unlike us, they did not follow a labeled taxonomy and considered that a repair action is any combination of a change operator and a code entity (e.g., update expression). It is worth mentioning that our two papers published on this subject (SOBREIRA et al., 2018; MADEIRAL et al., 2018) were published in the same year or just before the work presented by Liu et al. (2018a).

## 4.4 Final Remarks

In this chapter, we presented a descriptive study on BEARS, Bugs.jar, and Defects4J. This study includes the collection of information on bugs according to three aspects: the projects from which the bugs were mined, their exposure by failing test cases, and the patches written by developers for them. To collect information on the developer-written

patches, we developed the ADD tool. While presenting the results, we also put some findings from the study in the context of existing repair approaches and tools when they were applied based on our knowledge of the existing work.

We observed several differences and similarities among the three benchmarks of bugs. First, in the analysis of projects, we found that Defects4J is the only benchmark that does not contain bugs from multi-module projects. This has a big implication in the evaluation of repair tools because Defects4J is the benchmark adopted by the repair community, and multi-module architecture has become common for developing software systems nowadays. Considering the exposure of bugs by failing test cases, we found that all benchmarks follow the same patterns, e.g., the majority of their bugs are exposed by single test cases, bugs exposed by test failures are more frequent than bugs exposed by tests in error, and their exception type rankings are similar. Regarding patches, we found that the ones from Defects4J are smaller and less spread than the ones in Bears and Bugs.jar, and the top most performed repair actions and patterns by developers are the same for the three benchmarks. Finally, we make the observation that Defects4J differs from Bears and Bugs.jar in several aspects, which suggests that the extensive and most of the times exclusive usage of Defects4J for evaluating the effectiveness of repair tools should be avoided by conducting evaluations that also include other benchmarks of bugs.

Chapter 5

# An Empirical Study on Repair Tools versus Bug Benchmarks

Automatic program repair tools are used in *empirical evaluations* so that the repairability of the repair approaches they implement is measured. These evaluations consist of four main aspects in general: 1) [benchmark] the selection of benchmarks of bugs; 2) [execution] the collection of data by executing repair tools on the selected bugs; 3) [observed aspect] an investigation on the effectiveness of the repair approach regarding some criteria (e.g., *repairability*, *correctness*, and *repair time*); and finally 4) [comparison/discussion] the comparison of repair approaches and discussion.

A major threat of validity in existing evaluations, focusing on repair for Java programs, is that repair tools were widely evaluated on the same benchmark of bugs: Defects4J (JUST et al., 2014). This should not be a problem if Defects4J is not biased. However, no benchmark is perfect (LE GOUES et al., 2015), and the extent to which Defects4J is representative is unknown because even the distribution of real-world bugs is unknown. Therefore, by using a single benchmark when evaluating repair tools, a bias can be introduced, which makes it difficult to generalize the performance of repair tools.

In this chapter, we report on an experiment conducted on 11 test-suite-based repair tools for Java using benchmarks other than Defects4J. The primary goal of this experiment is to investigate if the existing repair tools have similar performance across different benchmarks of bugs. If a repair tool performs significantly better on one benchmark than on others, we say that the repair tool *overfits the benchmark*. The secondary goal is to understand the causes of non-patch generation from a practical view. To the best of our knowledge, our goals have not been the subject of investigation by the repair community.

The remainder of this chapter is organized as follows. Section 5.1 presents the design of our study, including research questions and data collection and analysis. Section 5.2 presents the results, followed by discussions in Section 5.3. Finally, Section 5.4 presents the final remarks.

## 5.1 Study Design

To achieve our goals, we designed our experiment considering the four main aspects usually used to evaluate repair tools: a) on benchmark, we use three benchmarks (including Defects4J), totaling 1,804 bugs; b) on execution, we run 11 repair tools on the 1,804 bugs; c) on the observed aspect, we analyze the repairability of the tools, focusing on their performance across the different benchmarks; and d) on comparison, we compare 11 repair tools.

### 5.1.1 Research Questions

In this study, we aim to answer the following research questions:

**RQ #1**. [Repairability] To what extent do test-suite-based repair tools generate patches for bugs from a diversity of benchmarks?

This research question guides us toward the investigation of the ability of the existing repair tools to generate test-suite adequate patches for bugs from the selected benchmarks.

**RQ #2**. [Benchmark overfitting] Is the repair tools' repairability similar across benchmarks?

The repair tools have been extensively evaluated on Defects4J. Our goal in this research question is to investigate if they repair bugs from other benchmarks to a similar extent as they repair bugs from Defects4J.

**RQ #3**. [Non-patch generation] What are the causes that lead repair attempts to not generate patches?

Existing evaluations focus on the successful cases, i.e., the bugs for which a given repair tool generated patches. However, to the best of our knowledge, there is no study that investigates the unsuccessful cases, i.e., repair attempts with no generated patches. Our goal in this research question is to find the causes of non-patch generation so that the repair community can focus on practical limitations and improve their repair tools.

### 5.1.2 Subject Repair Tools

To include a Java test-suite-based program repair tool in our study, it must meet the following four inclusion criteria:

*Criterion #1.* The repair tool must be publicly available. Our study involves the execution of repair tools, so tools that are not publicly available are excluded. We exclude

tools with this criterion when 1) the paper in which the tool was described does not include a link to the tool, 2) we cannot find the tool by searching it with Google, and 3) we received an answer by email from the authors of the tool explaining why the tool is not available (e.g., ELIXIR has a confidentiality issue) or no answer at all.

*Criterion #2.* The repair tool must be possible to run. Some tools are publicly available, but they are not possible to run for diverse issues (e.g., ACS uses GitHub, which recently changed its interface and does not allow programmed queries).

*Criterion #3.* The repair tool must be possible to run on bugs from any benchmark of bugs. We cannot run tools on other benchmarks if they are hardcoded to specific ones (e.g., SimFix currently works only for Defects4J).

*Criterion #4.* The repair tool must require only the source code of the program under repair and its test suite used as an oracle. These two elements are the two inputs specified in the problem statement of test-suite-based automatic program repair (MONPERRUS, 2018a).

After checking all 24 repair tools presented in Table 3, we found 12 tools that meet the inclusion criteria outlined. One of them, ssFix, was further excluded because we had issues running it, so we ended up with 11 repair tools for our experiment. Table 18 presents the excluded and included tools, and for the excluded ones, it also shows the criterion they did not meet. Note that, among the included tools, we have eight generate-and-validate tools, the two semantics-driven tools, and the only metaprogramming-based tool. Therefore, we cover the three families of techniques. We briefly described each selected repair tool in Chapter 2, Section 2.1.2.

### 5.1.3 Subject Benchmarks of Bugs

To select benchmarks of bugs for our study, we used the same criteria as in the study presented in Chapter 4, Section 4.2.2, which results in BEARS, Bugs.jar, and Defects4J.

### 5.1.4 Data Collection and Analysis

To answer our research questions, we executed the 11 repair tools on the three benchmarks using the REPAIRTHEMALL framework[13], resulting in patches that are further used for analysis. In this section, we describe the setup of the repair tools and their execution, and the analysis we performed on the repair attempts to determine the possible causes of non-patch generation.

[13] The REPAIRTHEMALL framework has been developed by a collaborator of the author of this thesis to automate and simplify the execution of repair tools on different benchmarks. This framework is not a contribution of this thesis.

Table 18 – Selected repair tools based on our inclusion criteria.

| | Criteria | Repair tools |
|---|---|---|
| Excluded (13) | Not public (C1) | Elixir (Saha et al., 2017), PAR (Kim et al., 2013), SOFix (Liu; Zhong, 2018), xPAR (Le et al., 2016) |
| | Not working (C2) | ACS (Xiong et al., 2017), CapGen (Wen et al., 2018), DeepRepair (White et al., 2019) |
| | Only compatible with Defects4J (C3) | LSRepair (Liu et al., 2018b), SimFix (Jiang et al., 2018) |
| | Faulty class/method required (C4) | HDRepair (Le et al., 2016), Jaid (Chen et al., 2017), SketchFix (Hua et al., 2018) |
| | Others | ssFix (Xin; Reiss, 2017b) |
| Included (11) | ARJA (Yuan; Banzhaf, 2018), Cardumen (Martinez; Monperrus, 2018), DynaMoth (Durieux; Monperrus, 2016a), jGenProg (Martinez; Monperrus, 2016), GenProg-A (Yuan; Banzhaf, 2018), jKali (Martinez; Monperrus, 2016), Kali-A (Yuan; Banzhaf, 2018), jMutRepair (Martinez; Monperrus, 2016), Nopol (Xuan et al., 2016), NPEFix (Durieux et al., 2017), RSRepair-A (Yuan; Banzhaf, 2018) | |

Table 19 – The used version of each repair tool.

| Repair tool | Framework name | GitHub repository | Commit id |
|---|---|---|---|
| ARJA, GenProg-A, Kali-A, RSRepair-A | Arja | yyxhdy/arja | e60b990f9 |
| Cardumen, jGenProg, jKali, jMutRepair | Astor | SpoonLabs/astor | 26ee3dfc8 |
| DynaMoth, Nopol | Nopol | SpoonLabs/nopol | 7ba58a78d |
| NPEFix | – | Spirals-Team/npefix | 403445b9a |

#### 5.1.4.1 Repair Tools' Setup

For this experiment, we set the time budget to two hours per repair attempt: a repair attempt consists of the execution of one repair tool over one bug. We also configured the repair tools to stop the execution of repair attempts when they already generated one patch. However, ARJA, GenProg-A, Kali-A, RSRepair-A, and NPEFix do not have this option, they stop their repair attempts when they consume their own tentative budget, or by timeout. Moreover, we configured repair tools to run on one predefined random seed: due to the huge computational power required for this experiment, we were not able to run the repair tools with additional seeds. Finally, Table 19 presents the version of each repair tool that we used in this study.

#### 5.1.4.2 Large-scale Execution

To our knowledge, our experimental setup is the largest in patch generation studies in terms of the number of repair tools and bugs, and also in execution time. In total, we

executed 11 repair tools on 1,804 bugs from 85 open-source projects that the selected three benchmarks provide. This represents 19,844 repair attempts, which took 283 days and 12 hours of combined execution, almost a year of continuous execution. This experiment would not be possible without the support of the cluster Grid'5000 (BALOUEK et al., 2013) that provided us with the computing power required to conduct this work.

#### 5.1.4.3 Finding Causes of Non-patch Generation

Prior studies on patch generation mainly focused on the ability of approaches to generate patches and did not investigate the reasons why non-patch generation occurs. The study on non-patch generation is important so that authors of repair tools can improve their tools. Since there is a lack of knowledge on that subject, we are not able to automatically detect the reasons why patches are not generated for bugs, and therefore manual analysis is required. Due to the scale of our experiment setup including 19,844 patch generation attempts, it is unrealistic to manually analyze each attempt log to understand what happened. We identified the major causes of non-patch generation by analyzing a sample of the repair attempt logs. We did not predefine the sample because we observed during preliminary investigations that identical behaviors occur for groups of repair attempts. For instance, we found that for all bugs from a specific project, all repair tools had the same issue in the fault localization. For that reason, predefining a sample is not optimal because we would analyze repair attempt logs that we already know the cause of non-patch generation.

## 5.2 Results

The results of our empirical study, as well as the answers to our research questions, are presented in this section.

### 5.2.1 Repairability of the 11 Repair Tools (RQ #1)

In this research question, we analyze the repairability of the repair tools considering the number of patched bugs and the number of bugs that are uniquely patched by tools.

Figure 10 presents the repairability of the 11 repair tools in descending order by the total number of patched bugs. For each tool, it shows the number of unique bugs patched by the tool (dark grey), the number of patched bugs that other repair tools also patched (light grey), and the total patched bugs with the proportion over all 1,804 included in this study. For example, DynaMoth synthesizes patches for 198 bugs in total (9.2% of all bugs), where 74 are uniquely patched by DynaMoth, and 124 are patched by DynaMoth and other tools.

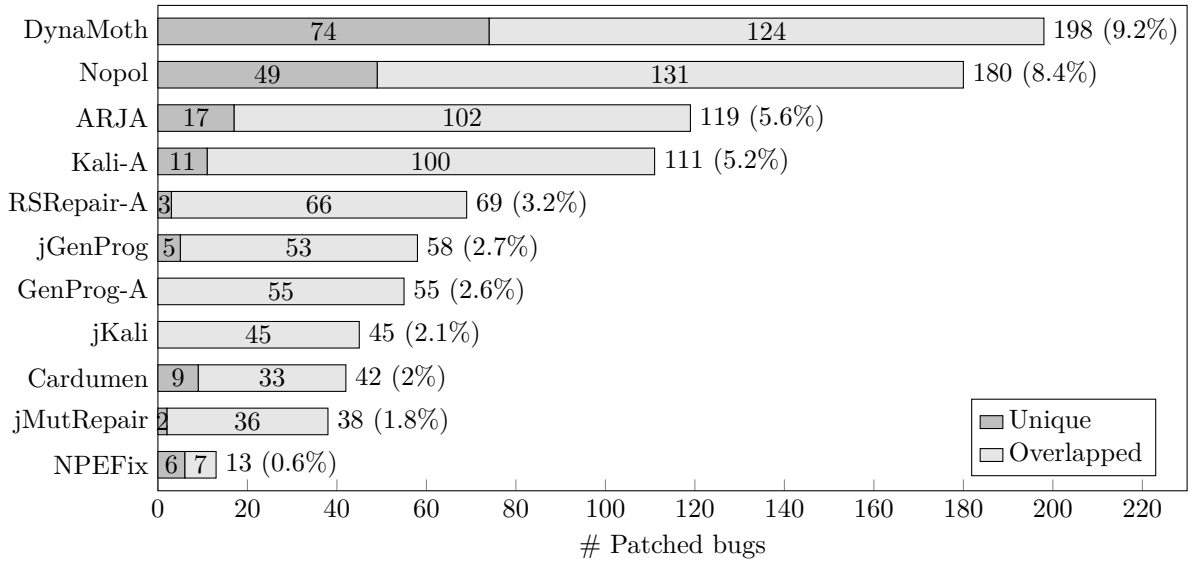


Figure 10 – Repairability of the 11 repair tools on 1,804 bugs.

We observe that DynaMoth, Nopol, and ARJA are the three tools that generate test-suite adequate patches for the highest number of bugs, with respectively 198, 180, and 119 patched bugs in total. NPEFix, on the other hand, generates patches for the fewest number of bugs (13). It can be explained by the narrow repair scope of this tool, i.e., bugs exposed by null pointer exception.

On bugs uniquely patched by tools, we observe that only GenProg-A and jKali did not succeed in generating patches for bugs that are not patched by other tools and that DynaMoth is the tool that patched the highest number of unique bugs. However, NPEFix is the tool that has the highest proportion of unique patched bugs, i.e., ∼50% of the 13 bugs patched by NPEFix are unique. These results show that the repair tools are complementary, e.g., each tool is able to generate patches for bugs that the other tools are not able to (except for GenProg-A and jKali).

The overlap between each pair of repair tools is presented in Table 20. In the case where the column name and the row name are the same (main diagonal), it presents the number of unique bugs patched by the tool. For instance, 17 bugs have been uniquely patched by ARJA, which repairs other 50 bugs that are also patched by GenProg-A.

We observe a large overlap between repair tools that share the same patch generation framework, i.e., the framework where the repair tools are implemented (see Table 19). For instance, ARJA has an overlap of 42% with GenProg-A, 63% with Kali-A, and 51% with RSRepair-A, all implemented in the Arja framework. However, ARJA has an overlap ranging from 3% to 36% with the other repair tools. DynaMoth has an overlap of 55% with Nopol, but only 0% to 24% with the other tools. Each tool implemented in the Astor framework has a large overlap with other tools in Astor. Moreover, the tools in Astor also have a high overlap with the tools in the Arja framework, which are tools that share similar repair approaches.

Table 20 – The overlap of the repair tools. Each row $r$ presents the percentage of bugs patched by the tool $t_r$ that were also patched by the rest of the tools. For instance, 42% of the bugs patched by ARJA (row 1) are also patched by GenProg-A (column 2). On the contrary, 91% of the bugs patched by GenProg-A (row 2) are also patched by ARJA (column 1).

| | ARJA | GenProg-A | Kali-A | RSRepair-A | Cardumen | jGenProg | jKali | jMutRepair | Nopol | DynaMoth | NPEFix |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ARJA | **14% (17)** | 42% (50) | 63% (75) | 51% (61) | 18% (22) | 33% (39) | 28% (33) | 21% (25) | 36% (43) | 34% (41) | 3% (4) |
| GenProg-A | 91% (50) | **0% (0)** | 78% (43) | 84% (46) | 29% (16) | 47% (26) | 42% (23) | 33% (18) | 49% (27) | 45% (25) | 4% (2) |
| Kali-A | 68% (75) | 39% (43) | **10% (11)** | 43% (48) | 17% (19) | 28% (31) | 33% (37) | 24% (27) | 46% (51) | 42% (47) | 2% (2) |
| RSRepair-A | 88% (61) | 67% (46) | 70% (48) | **4% (3)** | 23% (16) | 45% (31) | 33% (23) | 26% (18) | 41% (28) | 41% (28) | 3% (2) |
| Cardumen | 52% (22) | 38% (16) | 45% (19) | 38% (16) | **21% (9)** | 69% (29) | 48% (20) | 36% (15) | 24% (10) | 26% (11) | 5% (2) |
| jGenProg | 67% (39) | 45% (26) | 53% (31) | 53% (31) | 50% (29) | **9% (5)** | 57% (33) | 47% (27) | 31% (18) | 36% (21) | 3% (2) |
| jKali | 73% (33) | 51% (23) | 82% (37) | 51% (23) | 44% (20) | 73% (33) | **0% (0)** | 64% (29) | 51% (23) | 62% (28) | 2% (1) |
| jMutRepair | 66% (25) | 47% (18) | 71% (27) | 47% (18) | 39% (15) | 71% (27) | 76% (29) | **5% (2)** | 50% (19) | 50% (19) | 3% (1) |
| Nopol | 24% (43) | 15% (27) | 28% (51) | 16% (28) | 6% (10) | 10% (18) | 13% (23) | 11% (19) | **27% (49)** | 61% (109) | 1% (2) |
| DynaMoth | 21% (41) | 13% (25) | 24% (47) | 14% (28) | 6% (11) | 11% (21) | 14% (28) | 10% (19) | 55% (109) | **37% (74)** | 0% (1) |
| NPEFix | 31% (4) | 15% (2) | 15% (2) | 15% (2) | 15% (2) | 15% (2) | 8% (1) | 8% (1) | 15% (2) | 8% (1) | **46% (6)** |

**Answer to RQ #1**. **To what extent do test-suite-based repair tools generate patches for bugs from a diversity of benchmarks?** The 11 repair tools are able to generate patches for bugs ranging from 13 to 198 bugs, from a total of 1,804 bugs. They are complementary to each other because 9/11 repair tools fix unique bugs. We also observe that the overlapped repairability of the tools is impacted by their similar implemented repair approaches and also by the patch generation framework where they are implemented.

### 5.2.2 Benchmark Overfitting (RQ #2)

In this research question, we compare the repairability of the repair tools on the bugs of the extensively used benchmark Defects4J with their repairability on the other benchmarks included in this study (Bears and Bugs.jar).

Table 21 shows the number of bugs that have been patched by each repair tool per benchmark. We first observe that Defects4J is the benchmark with the highest number of unique patched bugs (185), which represents 46% of all Defects4J bugs. This difference can also be observed in the total number of generated patches per benchmark: Defects4J is still dominating the ranking with 550 generated patches, even though it contains fewer bugs than Bugs.jar (395 versus 1,158 bugs).

To test if the repairability of the repair tools (i.e., patched and non-patched bugs) is independent of Defects4J, we applied the Chi-square test on the number of patched bugs for Defects4J compared to the other benchmarks. The null hypothesis of our test is that *the number of patched bugs by a given tool is independent of Defects4J*. We observe in Table 21 that the *p-value* is smaller than the significance level $\alpha < .05$ for all repair tools.

Table 21 – The number of patched bugs per repair tool per benchmark. The last column presents the p-value of the Chi-square test of independence between the number of patched bugs from Defects4J compared to the other benchmarks.

| Repair tool \ Benchmark | Bears (251) | | Bugs.jar (1,158) | | Defects4J (395) | | Total (1,804) | | p-value |
|---|---|---|---|---|---|---|---|---|---|
| ARJA | 12 | (4%) | 21 | (1%) | 86 | (21%) | 119 | (6%) | < .00001 |
| GenProg-A | 1 | (<1%) | 9 | (<1%) | 45 | (11%) | 55 | (3%) | < .00001 |
| Kali-A | 15 | (5%) | 24 | (2%) | 72 | (18%) | 111 | (6%) | < .00001 |
| RSRepair-A | 1 | (<1%) | 6 | (<1%) | 62 | (15%) | 69 | (3%) | < .00001 |
| Cardumen | 13 | (5%) | 12 | (1%) | 17 | (4%) | 42 | (2%) | .003216 |
| jGenProg | 13 | (5%) | 14 | (1%) | 31 | (7%) | 58 | (3%) | < .00001 |
| jKali | 10 | (3%) | 8 | (<1%) | 27 | (6%) | 45 | (2%) | < .00001 |
| jMutRepair | 7 | (2%) | 11 | (<1%) | 20 | (5%) | 38 | (2%) | < .00001 |
| Nopol | 1 | (<1%) | 72 | (6%) | 107 | (27%) | 180 | (9%) | < .00001 |
| DynaMoth | 0 | (0%) | 124 | (10%) | 74 | (18%) | 198 | (10%) | < .00001 |
| NPEFix | 1 | (<1%) | 3 | (<1%) | 9 | (2%) | 13 | (<1%) | .000034 |
| Total | 74 | | 304 | | 550 | | 928 | | |
| Total unique | 25 | (9%) | 175 | (15%) | 185 | (46%) | 385 | (21%) | |

Hence, we reject the null hypothesis for these tools, and we conclude that the number of patched bugs by them is dependent of Defects4J. Therefore, *repair tools overfit Defects4J.*

The repairability of the repair tools on Defects4J cannot be only explained by the repair approaches. We raise three hypotheses that can explain the repairability difference between Defects4J and the other benchmarks: 1) there is a technical problem in the repair tools, 2) the bug fix isolation performed on Defects4J has an impact on the repair of Defects4J bugs, and 3) the distribution of the bug types in Defects4J is different from the other benchmarks.

**1.** [**Technical problems in the repair tools**] In RQ1, we observed the importance of the implementation of the tools for repairability. One hypothesis that can explain the fact that repair tools overfit on Defects4J is that the authors of the repair tools have debugged and tuned their frameworks for Defects4J and, consequently, improved significantly the repairability of their tools for this specific benchmark. For instance, they may have paid attention to not letting the dependencies of the repair tools interfere with the classpath of the Defects4J bugs, in order to preserve the behavior of test executions on the Defects4J bugs. However, this issue can affect the bugs of other benchmarks.

**2.** [**Bug fix isolation performed on Defects4J**] The second hypothesis is related to the way that Defects4J has been created. A bug fixing commit might include other changes that are not related to the actual bug fix. Then given a bug fixing

commit, the authors of Defects4J recreated the buggy and patched program versions so that the diff between the two versions contains only changes related to the bug fix: this is called bug fix isolation. The resulting isolated bug fixes facilitate studies on patches (SOBREIRA et al., 2018). However, such a procedure can potentially have an impact on the repairability of the repair tools. For instance, by comparing the developer patch[14] with the Defects4J patch[15] for the bug Closure-51, we observe that the method `isNegativeZero` has been introduced in the buggy program version, which contains part of the logic for fixing the bug. The presence of this method in the buggy program version can simplify the generation of patches by repair tools or introduce an ingredient for genetic programming repair approaches.

**3. [Bug type distribution in the benchmarks]** Our final hypothesis is related to the distribution of the bugs in the different benchmarks. Defects4J might contain more bugs that can be patched by the repair tools compared to the other benchmarks. For that reason, the bug type distribution of each benchmark should be further analyzed and correlated with the repairability of the tools.

To our understanding, the first hypothesis is more plausible since we observe in RQ1 that the implementation of the repair tools has an impact on their repairability. However, additional studies should be designed to identify which hypothesis, or a combination of hypotheses, has an impact on the repairability of the repair tools on Defects4J compared to the other benchmarks.

**Answer to RQ #2. Is the repair tools' repairability similar across benchmarks?** There is a difference in the repairability of the 11 repair tools across the benchmarks. The repairability of all tools is significantly higher for Defects4J bugs compared to the other benchmarks, so we conclude that they overfit Defects4J. In addition, we raised three hypotheses that could explain this difference. The confirmation of these hypotheses are full contributions themselves, which means that our study opens the opportunity for several future investigations.

### 5.2.3 Causes of Non-patch Generation (RQ #3)

In this final research question, we analyze the repair attempts that did not result in patches and identify the causes of non-patch generation. The goal of this research question is to provide insights to the automatic repair community on the causes of non-patch generation so that authors of repair tools can improve their tools.

[14] Human patch for Defects4J Closure-51 bug: <https://github.com/google/closure-compiler/commit/a02241e5df48e44e23dc0e66dbef3fdc3c91eb3e>.

[15] Defects4J patch for Closure-51 bug: <http://program-repair.org/defects4j-dissection/#!/bug/Closure/51>.

Table 22 – The percentage of repair attempts that failed by error.

| Repair tool | Bears | Bugs.jar | Defects4J | Average |
|---|---|---|---|---|
| ARJA | 24.7 | 49.56 | 1.26 | 35.53 |
| GenProg-A | 88.04 | 78.06 | 7.08 | 63.91 |
| Kali-A | 24.7 | 50.08 | 4.81 | 36.64 |
| RSRepair-A | 87.25 | 79.27 | 6.83 | 64.52 |
| Cardumen | 47.41 | 70.46 | 48.6 | 62.47 |
| jGenProg | 45.01 | 63.29 | 12.65 | 49.66 |
| jKali | 44.62 | 64.42 | 12.4 | 50.27 |
| jMutRepair | 72.11 | 66.66 | 15.44 | 56.2 |
| Nopol | 28.68 | 60.27 | 45.31 | 52.6 |
| DynaMoth | 27.09 | 46.97 | 4.3 | 34.86 |
| NPEFix | 89.24 | 86.18 | 73.16 | 83.75 |
| Average | 52.62 | 65.02 | 21.08 | 53.67 |

Table 23 – The percentage of repair attempts that failed by timeout.

| Repair tool | Bears | Bugs.jar | Defects4J | Average |
|---|---|---|---|---|
| ARJA | 19.52 | 18.56 | 6.07 | 15.96 |
| GenProg-A | 6.37 | 7.08 | 9.62 | 7.53 |
| Kali-A | 1.19 | 2.76 | 0.0 | 1.94 |
| RSRepair-A | 7.17 | 6.99 | 8.86 | 7.42 |
| Cardumen | 4.38 | 61.57 | 19.74 | 44.45 |
| jGenProg | 48.2 | 28.23 | 71.39 | 40.46 |
| jKali | 0.79 | 4.05 | 1.77 | 3.1 |
| jMutRepair | 0.39 | 3.45 | 1.01 | 2.49 |
| Nopol | 0.39 | 0.51 | 0.0 | 0.38 |
| DynaMoth | 0.0 | 0.69 | 2.27 | 0.94 |
| NPEFix | 0.0 | 4.49 | 0.75 | 3.04 |
| Average | 8.04 | 12.58 | 11.04 | 11.61 |

Table 22 and Table 23 present the percentage of repair attempts that finished due to an error and by timeout, respectively. They show that repair attempts in error or timeout represent the majority of all repair attempts (∼65%). The Bugs.jar benchmark is the main contributor to this percentage. The size and complexity of the Bugs.jar projects show the limitation of the current automatic patch generation tools.

Moreover, Table 22 shows that NPEFix is the tool with the highest error rate, but this tool crashes when no null pointer exception is found in the execution of the failing test case that exposes a bug. On the other hand, DynaMoth is a more reliable tool. Regarding the timeout in Table 23, Cardumen and jGenProg are more susceptible to reaching the timeout.

We then manually analyzed the execution trace of the repair attempts to identify the causes of non-patch generation. The methodology for this analysis is described in Section 5.1.4.3, and by following it we identified six causes of non-patch generation.

**1.** [**The repair tool cannot repair the bug**] A logical problem is that the repair tools do not have a patch that fixes the bug in their search space. For instance, NPEFix is not able to generate patches for bugs that are not related to null-pointer exceptions. jGenProg is not able to generate a patch when the repair ingredient is not in the source code of the application, which occurs frequently for small programs. New repair approaches should be created to handle this cause of non-patch generation.

**2.** [**Incorrect fault localization**] When the fault localization does not succeed in identifying the location of the bug, the repair tools do not succeed in generating a patch for it. This can be due to a limitation of the fault localization approach or to the suspiciousness threshold that the repair tools use. Moreover, we identified that test cases that should pass are failing, and consequently there is a misleading fault localization. For instance, the fault localization fails on all bugs from the INRIA/spoon project (from the Bears benchmark) because the fault localization does not succeed in loading a test resource and consequently the passing test cases fail.

**3.** [**Multiple fault locations**] Developers frequently fix a bug at more than one location in the source code: we refer to this type of bug as a multi-location bug. However, most current repair tools and fault localization tools do not support multi-location bugs. For instance, the bug Math-1 from Defects4J has to be fixed in the exact same way at two different locations, and the two locations are specified by two failing test cases. The current tools consider that the two failing test cases specify the same bug at the same location and consequently do not succeed in fixing it.

**4.** [**Small time budget**] We detected many repair attempts that failed by timeout, i.e., they finished the execution by consuming their time budget. It is not possible to predict the outcome of these attempts. In our experiment, the repair tools require 15 minutes on average to generate a patch, which is significantly lower than the allocated time budget (two hours). However, a previous study (MARTINEZ; MONPERRUS, 2018) showed that an additional time budget could result in a higher number of patches with genetic programming approaches. More studies should be conducted to investigate if drastically increasing the time budget makes the repair tools generate more patches.

**5.** [**Incorrect configuration**] We observed that the RepairThemAll framework does not succeed in correctly computing parameters for some bugs to give as input to the repair tools, such as compliance level, source folders, and failing test cases. This results in failing repair attempts, which can be due to a bug in RepairThemAll or an impossibility of compiling the bug.

**6.** [**Other technical issues**] The final cause of non-patch generation is related to other technical limitations that cause the non-execution of the repair tools. One of them is about too long command lines. The repair tools are executed from the

command line, which means that all parameters must be provided on the command line. However, the size of the command line is limited and in the case of projects that have a long classpath, the operating system denies the execution of the command line, which results in failing repair attempts. On Bugs.jar, for instance, many repair attempts finished with the error `[Errno 7] Argument list too long`. Finally, there are also other diverse issues that cause the repair tools to crash. For instance, jGenProg finished its repair attempt on the bug Flink-6bc6dbec from Bugs.jar with a `NullPointerException`.

**Answer to RQ #3. What are the causes that lead repair attempts to not generate patches?** Through an analysis of logs of repair attempts, we identified six causes of non-patch generation, such as incorrect fault localization. Each cause should be investigated in detail in new studies. Moreover, repair tools' designers are also stakeholders of those causes, which inform them what are the weaknesses of their tools and help them to understand their previous evaluations' results.

## 5.3 Discussion

### 5.3.1 Impact of the Repair Tools' Engineering on Repairability

During the execution of this study, we observed that the implementations of the repair approaches play an important role in their ability to repair bugs. For instance, jKali and Kali-A share the same approach but neither have the same implementation nor the same results (see Table 21). Kali-A fixes 111 bugs while jKali fixes 45 with the same input. Note that this observation has also been correlated with the analysis of non-patch generation, where a significant number of causes is not related to the repair approaches themselves, but to their implementations.

This observation highlights a potential bias in empirical studies on automatic program repair that compare the repairability of different repair approaches. Based on this observation, these studies only compare the effectiveness of repair tools, not the approaches themselves. Hence, no comparison can be reliably made to compare repair approaches.

### 5.3.2 The Observed Repairability Compared to the Previous Evaluations

Table 3 shows the test-suite-based repair tools for Java and the repairability results from their previous evaluations. It is difficult to compare those results with ours because the previous evaluations on Defects4J did not consider all bugs of the benchmark. On Defects4J, only Cardumen fixes fewer bugs in this study compared to the previous evalu-

ation. This can be explained by the setup difference (such as the number of random seeds considered in the study) and potential bugs in the version of Cardumen we used.

### 5.3.3 Threats to Validity

This study focuses on test-suite adequate patches, which means that the generated patches make the test suite pass; yet, there is no guarantee that they fix the bugs. Studying patch correctness (XIN; REISS, 2017a; YU et al., 2019) is out of the scope of this study. Our goal is to analyze the current state of the automatic program repair tools and to identify potential flaws and improvements to be made. The conclusions of our study do not require knowledge on the correctness of the patches.

Our goal is to have a complete picture of test-suite-based repair tools for Java. In our literature review, presented in Chapter 2, Table 1, we found 24 repair tools that compose the complete picture. Our study was conducted considering only 11 of them: note that this is the largest experiment in terms of the number of repair tools (and benchmarks). However, we do not have the full picture we wanted, which is a threat to the external validity of our results. Most of the repair tools that we did not include in our study are simply not possible to run: for instance, PAR (KIM et al., 2013) is not even available. Open-source tools allow the community to build knowledge in several directions. In our work, open-source tools allowed us to perform a novel evaluation of the state of the repair tools. Another direction is to help the development of new tools: for instance, DeepRepair (WHITE et al., 2019) is built on Astor (MARTINEZ; MONPERRUS, 2016), which is a library for repairing.

### 5.3.4 Related Work

The works related to ours are empirical studies on the *repairability* of multiple automatic program repair for Java programs.

Martinez et al. (2017) reported on a remarkably large experiment, where three repair tools (jGenProg, jKali, and Nopol) were executed on Defects4J. The focus of their study was to measure the repairability of repair tools and find correct patches by manual analysis. They found that a small number of bugs (9/47) could be repaired with a test-suite adequate patch that is also correct.

Motwani et al. (2018) reported on an empirical study that included seven repair tools for the Java and C programming languages, where the Defects4J and ManyBugs benchmarks were used. They had a different focus and investigated if bugs repaired by repair tools are hard and important. To do so, they used the repairability data from previous works and performed a correlation analysis between the repaired bugs and measures of defect importance, human-written patch complexity, and test suite quality.

Ye et al. (2019) presented a study in which nine repair tools were executed on QuixBugs. They used automatically generated test cases based on human-written patches to identify incorrect patches generated by the repair tools. The goal and scale of our study differ from those of Ye et al. We performed a large-scale experiment to investigate benchmark overfitting and non-patch generation, not to detect overfitting patches.

## 5.4 Final Remarks

In this chapter, we presented an empirical study including 11 repair tools and 1,804 bugs from three benchmarks. In total, 19,844 repair attempts were made. This is the largest experiment in the field to the best of our knowledge. The goal of our experiment is to obtain an overview of the current state of repair tools for Java in practice. For that, we scaled up the previous experiments by considering more benchmarks of bugs, which combined have bugs from 85 projects, collected with different strategies. Then, we were able to investigate *the benchmark overfitting* problem.

We found that repair tools are able to repair bugs from benchmarks that were not initially used for their evaluations. However, our results suggest that all repair tools overfit Defects4J. Finally, we analyzed why repair tools do not succeed in generating patches, which resulted in six different causes that can help the future development of repair tools.

Chapter 6

# Conclusion

The *automatic program repair* research field was activated in 2009, with the seminal repair approach GenProg (WEIMER et al., 2009). In the last decade, several approaches and tools were proposed to automatically repair bugs. However, despite the efforts of researchers in the last few years, there is still a long road towards the usage of automatic program repair tools in practice. In this thesis, we presented contributions to the automatic program repair research field focusing on the *evaluation* of repair tools, where *benchmarks of bugs* play an important role.

We first addressed the problem of the scarcity of benchmarks of bugs. For that, we built the BEARS benchmark (Chapter 3). Our contribution includes 1) an original approach to collect bugs and their patches based on commit building state from Continuous Integration, 2) a tool, named BEARS-COLLECTOR, which implements our approach, and 3) a benchmark of bugs, named BEARS-BENCHMARK, which contains 251 bugs from 72 projects. To our knowledge, it is the largest benchmark of reproducible bugs with respect to project diversity (the closest benchmark is Bugs.jar, which covers only eight projects).

Second, we addressed the problem of the lack of knowledge on benchmarks of bugs. We presented a descriptive study on BEARS and the state-of-the-art Defects4J and Bugs.jar (Chapter 4). Our contribution includes 1) an analysis of three benchmarks, and 2) a tool, named ADD, which extracts properties on patches for bugs.

Finally, we addressed the problem of the extensive usage of Defects4J for evaluating automatic program repair tools. We conducted an empirical study on repair tools on different benchmarks of bugs, where we investigated *the benchmark overfitting* problem. Our contribution includes 1) the first study on the repairability of repair tools across multiple benchmarks, where the goal was to investigate if the repair tools perform significantly better on the extensively used benchmark Defects4J than on other benchmarks, and 2) a thorough study on the non-patch generation cases.

| Our findings suggest that the benchmarks of bugs are complementary to each other and that the usage of multiple and diverse benchmarks of bugs is key to evaluating the generalization of the effectiveness of automatic repair tools. |
|---|

## 6.1 Summary of Artifacts

Several artifacts were produced during this work, such as tools and datasets. We have worked to make them available toward open science. Table 24 summarizes the artifacts, where the ones marked with "⋆" are related but not claimed as contributions from this thesis. All data-related artifacts have a website to present data.

Table 24 – Summary of artifacts.

| GitHub repository | Type | Role | Description |
|---|---|---|---|
| bears-bugs/bears-benchmark | Dataset | Leader | BEARS-BENCHMARK |
| lascam-UFU/automatic-diff-dissection | Tool | Leader | ADD |
| ⋆ program-repair/defects4j-dissection | Data | Collaborator | Defects4J analysis |
| ⋆ program-repair/RepairThemAll | Tool | Collaborator | Benchmark overfitting study |
| program-repair/RepairThemAll_experiment | Data | Collaborator | Benchmark overfitting study |

## 6.2 Bibliographical Contributions

The main chapters of this thesis contain content from the following papers, of which I am the main author or contributed equally as the first author:

- Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. BEARS*: An Extensible Java Bug Benchmark for Automatic Program Repair Studies*. Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '19), pages 468–478.

- Fernanda Madeiral, Thomas Durieux, Victor Sobreira, and Marcelo Maia. *Towards an automated approach for bug fix pattern detection*. Proceedings of the VI Workshop on Software Visualization, Evolution and Maintenance (VEM '18), pages 163–170. (Best Paper Award)

- Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. *Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts*. Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), pages 302–313.

The following paper is highly related to this thesis but just used as background:

- Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. *Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J*. Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '18), pages 130–140.

Moreover, during the three first years of Ph.D., I worked on a different Software Engineering topic and also collaborated with other researchers. Appendix B presents bibliographical contributions that are not related to this thesis.

## 6.3 Future Work

There are several future works that can improve or extend the contributions of this thesis. We group these works by the main subjects: Bears, benchmark studies, and repair tool studies.

**Bears.** The major drawback of the Bears project is the final manual analysis of the collected branches. The benchmarks related to Bears, which are Defects4J and Bugs.jar, were also constructed with approaches that include manual analysis. However, due to the high automation of Bears to collect bugs from a diversity of projects, which performs attempts to reproduce test failures, a lot of false positives are obtained. In this context, one future work is to create heuristics to discard Bears branches that do not look like to contain a bug fix (e.g., branches containing refactoring). This would greatly minimize the effort spent on manual validation.

The collection of bugs in real time performed by Bears opens the opportunity to contact the developers who have just fixed bugs. The developers are experts on the projects and also on the fixed bugs, i.e., they wrote the collected human-written patches. Contacting them just after bugs have been fixed would be an invaluable source of information about the bugs, as well as the bug fixing activity.

**Studies on benchmarks.** Looking at the descriptive study on benchmarks from a high-level view, there are several other aspects of the projects, failing tests, and patches that can be studied in order to obtain more information on the bugs and the composition of the benchmarks. For instance, the coverage of tests might bring valuable information on bugs. Moreover, bugs could be classified according to the target bug classes of repair tools, which would result in valuable information for their evaluations.

On the patch analysis, there are several improvements that can be performed. First, ADD has been evaluated only on Defects4J. Further evaluations using other benchmarks, such as Bears and Bugs.jar, should be conducted to increase the external validity of our evaluation and find opportunities to improve ADD. Second, ADD outputs a file containing the extracted properties. A visualization for patches, where repair patterns, for instance, are highlighted, would support the human patch comprehension task.

**Studies on repair tools.** Our empirical study on repair tools opens several opportunities for future investigations. First, our hypotheses about why repair tools perform better on Defects4J can be further confirmed. For instance, one of the hypotheses is the fact that the buggy program versions were changed in Defects4J due to the bug fix isolation process.

A study to confirm this hypothesis is a complete contribution itself, which would require the collection of the Defects4J bugs without the isolation of bug fixes and the execution of the repair tools on them.

Second, other repair tools can also be executed to scale up our study. ssFix, for instance, is possible to run, despite the fact that we had some issues for it, which led to its exclusion in this work. Moreover, the tools that are hardcoded to run on Defects4J could also be adapted to work on other benchmarks of bugs.

Finally, advanced studies correlating data from benchmarks and repair tools should be conducted. For instance, correlating repair tool results with characteristics of bugs such as those presented in our descriptive study on benchmarks could provide valuable insights into the effectiveness of repair tools.

## 6.4 Last Words

The contributions of this thesis are just small pieces toward progress in automatic program repair research. As last words, we would like to state that, as a belief, the *generalization* of repair tools' performance can only be measured on *unbiased benchmarks* of bugs that represent the population of projects and bugs in the real world well. *Maybe* we have unbiased benchmarks. *Maybe* we also have only biased benchmarks. The "*maybe*" is the problem: we do not know the existing population of projects and bugs in the world. Therefore, we do not know how an unbiased sample should be. According to Monperrus (2018a), "*there are many bug classes for which there are no clear definition and scope in the literature, and some of them even miss a name. [...] building a comprehensive taxonomy of bug classes will require years of research*". Ergo, there are still many studies to be conducted toward finding the population of projects and bugs in the world, which would support the foundation of benchmarks of bugs, and by extension reveal the generalization of automatic program repair tools.

# Bibliography

AVIZIENIS, A.; LAPRIE, J.-C.; RANDELL, B.; LANDWEHR, C. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, IEEE Computer Society Press, Washington, DC, USA, v. 1, n. 1, p. 11–33, jan. 2004. Available at: <https://doi.org/10.1109/TDSC.2004.2>.

BALOUEK, D.; AMARIE, A. C.; CHARRIER, G.; DESPREZ, F.; JEANNOT, E.; JEANVOINE, E.; LÈBRE, A.; MARGERY, D.; NICLAUSSE, N.; NUSSBAUM, L.; RICHARD, O.; PÉREZ, C.; QUESNEL, F.; ROHR, C.; SARZYNIEC, L. Adding Virtualization Capabilities to the Grid'5000 Testbed. In: IVANOV, I. I.; SINDEREN, M. van; LEYMANN, F.; SHAN, T. (Ed.). *Cloud Computing and Services Science (Communications in Computer and Information Science)*. Cham: Springer International Publishing, 2013. v. 367, p. 3–20. Available at: <https://doi.org/10.1007/978-3-319-04519-1_1>.

BELLER, M.; GOUSIOS, G.; ZAIDMAN, A. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In: *Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17)*. Piscataway, NJ, USA: IEEE Press, 2017. p. 447–450. Available at: <https://doi.org/10.1109/MSR.2017.24>.

CHEN, L.; PEI, Y.; FURIA, C. A. Contract-Based Program Repair without the Contracts. In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*. Piscataway, NJ, USA: IEEE Press, 2017. p. 637–647. Available at: <https://doi.org/10.1109/ASE.2017.8115674>.

DALLMEIER, V.; ZIMMERMANN, T. Extraction of Bug Localization Benchmarks from History. In: *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*. New York, NY, USA: ACM, 2007. p. 433–436. Available at: <https://doi.org/10.1145/1321631.1321702>.

DEBROY, V.; WONG, W. E. Using Mutation to Automatically Suggest Fixes for Faulty Programs. In: *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST '10)*. Washington, DC, USA: IEEE Computer Society, 2010. p. 65–74. Available at: <https://doi.org/10.1109/ICST.2010.66>.

DO, H.; ELBAUM, S.; ROTHERMEL, G. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering*, Kluwer Academic Publishers, Hingham, MA, USA, v. 10, n. 4, p. 405–435, out. 2005. Available at: <https://doi.org/10.1007/s10664-005-3861-2>.

DURIEUX, T.; CORNU, B.; SEINTURIER, L.; MONPERRUS, M. Dynamic Patch Generation for Null Pointer Exceptions Using Metaprogramming. In: *Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '17)*. Klagenfurt, Austria: IEEE, 2017. p. 349–358. Available at: <https://doi.org/10.1109/SANER.2017.7884635>.

DURIEUX, T.; MONPERRUS, M. DynaMoth: Dynamic Code Synthesis for Automatic Program Repair. In: *Proceedings of the 11th International Workshop on Automation of Software Test (AST '16)*. New York, NY, USA: ACM, 2016. p. 85–91. Available at: <https://doi.org/10.1145/2896921.2896931>.

DURIEUX, T.; MONPERRUS, M. *IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs*. University of Lille, 2016.

FALLERI, J.-R.; MORANDAT, F.; BLANC, X.; MARTINEZ, M.; MONPERRUS, M. Fine-grained and Accurate Source Code Differencing. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. New York, NY, USA: ACM, 2014. p. 313–324. Available at: <https://doi.org/10.1145/2642937.2642982>.

FLURI, B.; WUERSCH, M.; PINZGER, M.; GALL, H. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering*, IEEE Press, Piscataway, NJ, USA, v. 33, n. 11, p. 725–743, nov. 2007. Available at: <https://doi.org/10.1109/TSE.2007.70731>.

GAZZOLA, L.; MICUCCI, D.; MARIANI, L. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering*, IEEE, v. 45, n. 1, p. 34–67, jan. 2019. Available at: <https://doi.org/10.1109/TSE.2017.2755013>.

GOUSIOS, G. The GHTorrent Dataset and Tool Suite. In: *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. Piscataway, NJ, USA: IEEE Press, 2013. p. 233–236. Available at: <https://doi.org/10.1109/MSR.2013.6624034>.

HUA, J.; ZHANG, M.; WANG, K.; KHURSHID, S. Towards Practical Program Repair with On-Demand Candidate Generation. In: *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. New York, NY, USA: ACM, 2018. p. 12–23. Available at: <https://doi.org/10.1145/3180155.3180245>.

HUTCHINS, M.; FOSTER, H.; GORADIA, T.; OSTRAND, T. Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria. In: *Proceedings of the 16th International Conference on Software Engineering (ICSE '94)*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994. p. 191–200. Available at: <https://doi.org/10.1109/ICSE.1994.296778>.

JIANG, J.; XIONG, Y.; ZHANG, H.; GAO, Q.; CHEN, X. 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 '18)*. New York, NY, USA: ACM, 2018. p. 298–309. Available at: <https://doi.org/10.1145/3213846.3213871>.

JUST, R.; JALALI, D.; ERNST, M. D. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In: *Proceedings of the 23rd International Symposium on Software Testing and Analysis (ISSTA '14)*. New York, NY, USA: ACM, 2014. p. 437–440. Available at: <https://doi.org/10.1145/2610384.2628055>.

KIM, D.; NAM, J.; SONG, J.; KIM, S. Automatic Patch Generation Learned from Human-Written Patches. In: *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*. Piscataway, NJ, USA: IEEE Press, 2013. p. 802–811. Available at: <https://doi.org/10.1109/ICSE.2013.6606626>.

LANGR, J.; HUNT, A.; THOMAS, D. *Pragmatic Unit Testing in Java 8 with JUnit*. 1st. ed. : Pragmatic Bookshelf, 2015. ISBN 1941222595, 9781941222591.

LE GOUES, C.; FORREST, S.; WEIMER, W. Current Challenges in Automatic Software Repair. *Software Quality Journal*, Kluwer Academic Publishers, Hingham, MA, USA, v. 21, n. 3, p. 421–443, set. 2013. Available at: <https://doi.org/10.1007/s11219-013-9208-0>.

LE GOUES, C.; HOLTSCHULTE, N.; SMITH, E. K.; BRUN, Y.; DEVANBU, P.; FORREST, S.; WEIMER, W. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering*, IEEE Press, Piscataway, NJ, USA, v. 41, n. 12, p. 1236–1256, dez. 2015. Available at: <https://doi.org/10.1109/TSE.2015.2454513>.

LE GOUES, C.; NGUYEN, T.; FORREST, S.; WEIMER, W. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering*, IEEE Press, Piscataway, NJ, USA, v. 38, n. 1, p. 54–72, jan. 2012. Available at: <https://doi.org/10.1109/TSE.2011.104>.

LE, X. B. D.; LO, D.; LE GOUES, C. History Driven Program Repair. In: *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '16)*. Suita, Japan: IEEE, 2016. p. 213–224. Available at: <https://doi.org/10.1109/SANER.2016.76>.

LIN, D.; KOPPEL, J.; CHEN, A.; SOLAR-LEZAMA, A. QuixBugs: A Multi-Lingual Program Repair Benchmark Set Based on the Quixey Challenge. In: *Proceedings of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion 2017)*. New York, NY, USA: ACM, 2017. p. 55–56. Available at: <https://doi.org/10.1145/3135932.3135941>.

LIU, K.; KIM, D.; KOYUNCU, A.; LI, L.; BISSYANDÉ, T. F.; TRAON, Y. L. A Closer Look at Real-World Patches. In: *Proceedings of the 34th International Conference on Software Maintenance and Evolution (ICSME '18)*. IEEE Computer Society, 2018. p. 275–286. Available at: <https://doi.org/10.1109/ICSME.2018.00037>.

LIU, K.; KOYUNCU, A.; KIM, K.; KIM, D.; BISSYANDÉ, T. F. LSRepair: Live Search of Fix Ingredients for Automated Program Repair. In: *Proceedings of the 25th Asia-Pacific Software Engineering Conference (APSEC '18)*. Washington, DC, USA: IEEE Computer Society, 2018. p. 658–662. Available at: <https://doi.org/10.1109/APSEC.2018.00085>.

LIU, X.; ZHONG, H. Mining StackOverflow for Program Repair. In: *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '18)*. Campobasso, Italy: IEEE, 2018. p. 118–129. Available at: <https://doi.org/10.1109/SANER.2018.8330202>.

LIU, Y.; ZHANG, L.; ZHANG, Z. A Survey of Test Based Automatic Program Repair. *Journal of Software*, v. 13, n. 8, p. 437–452, ago. 2018. Available at: <https://doi.org/10.17706/jsw.13.8.437-452>.

LU, S.; LI, Z.; QIN, F.; TAN, L.; ZHOU, P.; ZHOU, Y. BugBench: Benchmarks for Evaluating Bug Detection Tools. In: *Proceedings of the Workshop on the Evaluation of Software Defect Detection Tools*. 2005.

LUO, Q.; HARIRI, F.; ELOUSSI, L.; MARINOV, D. An Empirical Analysis of Flaky Tests. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*. New York, NY, USA: ACM, 2014. p. 643–653. Available at: <https://doi.org/10.1145/2635868.2635920>.

MADEIRAL, F.; DURIEUX, T.; SOBREIRA, V.; MAIA, M. Towards an automated approach for bug fix pattern detection. In: *Proceedings of the VI Workshop on Software Visualization, Evolution and Maintenance (VEM '18)*. 2018. p. 163–170. Available at: <https://arxiv.org/abs/1807.11286>.

MARTINEZ, M.; DUCHIEN, L.; MONPERRUS, M. Automatically Extracting Instances of Code Change Patterns with AST Analysis. In: *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM '13)*. USA: IEEE Computer Society, 2013. p. 388–391. Available at: <https://doi.org/10.1109/ICSM.2013.54>.

MARTINEZ, M.; DURIEUX, T.; SOMMERARD, R.; XUAN, J.; MONPERRUS, M. Automatic Repair of Real Bugs in Java: A Large-scale Experiment on the Defects4J Dataset. *Empirical Software Engineering*, Kluwer Academic Publishers, Hingham, MA, USA, v. 22, n. 4, p. 1936–1964, ago. 2017. Available at: <https://doi.org/10.1007/s10664-016-9470-4>.

MARTINEZ, M.; MONPERRUS, M. ASTOR: A Program Repair Library for Java. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA '16), Demonstration Track*. New York, NY, USA: ACM, 2016. p. 441–444. Available at: <https://doi.org/10.1145/2931037.2948705>.

MARTINEZ, M.; MONPERRUS, M. Ultra-Large Repair Search Space with Automatically Mined Templates: the Cardumen Mode of Astor. In: COLANZI, T. E.; MCMINN, P. (Ed.). *Proceedings of the 10th International Symposium on Search-Based Software Engineering (SSBSE '18). Lecture Notes in Computer Science, vol 11036*. Cham: Springer International Publishing, 2018. p. 65–86. Available at: <https://doi.org/10.1007/978-3-319-99241-9_3>.

MARTINEZ, M.; WEIMER, W.; MONPERRUS, M. Do the Fix Ingredients Already Exist? An Empirical Inquiry into the Redundancy Assumptions of Program Repair Approaches. In: *Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014)*. New York, NY, USA: ACM, 2014. p. 492–495. Available at: <https://doi.org/10.1145/2591062.2591114>.

MONPERRUS, M. A Critical Review of "Automatic Patch Generation Learned from Human-Written Patches": Essay on the Problem Statement and the Evaluation of Automatic Software Repair. In: *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*. New York, NY, USA: ACM, 2014. p. 234–242. Available at: <https://doi.org/10.1145/2568225.2568324>.

MONPERRUS, M. Automatic Software Repair: A Bibliography. *ACM Computing Surveys*, ACM, New York, NY, USA, v. 51, n. 1, p. 17:1–17:24, jan. 2018. Available at: <https://doi.org/10.1145/3105906>.

MONPERRUS, M. *The Living Review on Automated Program Repair*. HAL/archives-ouvertes.fr, 2018.

MOTWANI, M.; SANKARANARAYANAN, S.; JUST, R.; BRUN, Y. Do automated program repair techniques repair hard and important bugs? *Empirical Software Engineering*, Springer US, v. 23, n. 5, p. 2901–2947, out. 2018. Available at: <https://doi.org/10.1007/s10664-017-9550-0>.

PAN, K.; KIM, S.; WHITEHEAD JR., E. J. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, Kluwer Academic Publishers, Hingham, MA, USA, v. 14, n. 3, p. 286–315, jun. 2009. Available at: <https://doi.org/10.1007/s10664-008-9077-5>.

PAWLAK, R.; MONPERRUS, M.; PETITPREZ, N.; NOGUERA, C.; SEINTURIER, L. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software–Practice & Experience*, John Wiley & Sons, Inc., New York, NY, USA, v. 46, n. 9, p. 1155–1179, set. 2016. Available at: <https://doi.org/10.1002/spe.2346>.

QI, Y.; MAO, X.; LEI, Y.; DAI, Z.; WANG, C. The Strength of Random Search on Automated Program Repair. In: *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*. New York, NY, USA: ACM, 2014. p. 254–265. Available at: <https://doi.org/10.1145/2568225.2568254>.

QI, Z.; LONG, F.; ACHOUR, S.; RINARD, M. An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems. In: *Proceedings of the 24th International Symposium on Software Testing and Analysis (ISSTA '15)*. New York, NY, USA: ACM, 2015. p. 24–36. Available at: <https://doi.org/10.1145/2771783.2771791>.

RAY, B.; POSNETT, D.; DEVANBU, P.; FILKOV, V. A Large-Scale Study of Programming Languages and Code Quality in GitHub. *Communications of the ACM*, ACM, New York, NY, USA, v. 60, n. 10, p. 91–100, set. 2017. Available at: <https://doi.org/10.1145/3126905>.

SAHA, R. K.; LYU, Y.; LAM, W.; YOSHIDA, H.; PRASAD, M. R. Bugs.jar: A Large-scale, Diverse Dataset of Real-world Java Bugs. In: *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*. New York, NY, USA: ACM, 2018. p. 10–13. Available at: <https://doi.org/10.1145/3196398.3196473>.

SAHA, R. K.; LYU, Y.; YOSHIDA, H.; PRASAD, M. R. ELIXIR: Effective Object-Oriented Program Repair. In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*. Piscataway, NJ, USA: IEEE Press, 2017. p. 648–659. Available at: <https://doi.org/10.1109/ASE.2017.8115675>.

SOBREIRA, V.; DURIEUX, T.; MADEIRAL, F.; MONPERRUS, M.; MAIA, M. A. Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J. In: *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '18)*. IEEE, 2018. p. 130–140. Available at: <https://doi.org/10.1109/SANER.2018.8330203>.

TAN, S. H.; YI, J.; YULIS; MECHTAEV, S.; ROYCHOUDHURY, A. Codeflaws: A Programming Competition Benchmark for Evaluating Automated Program Repair Tools. In: *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C '17)*. Piscataway, NJ, USA: IEEE Press, 2017. p. 180–182. Available at: <https://doi.org/10.1109/ICSE-C.2017.76>.

URLI, S.; YU, Z.; SEINTURIER, L.; MONPERRUS, M. How to Design a Program Repair Bot? Insights from the Repairnator Project. In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '18)*. New York, NY, USA: ACM, 2018. p. 95–104. Available at: <https://doi.org/10.1145/3183519.3183540>.

WEIMER, W.; NGUYEN, T.; LE GOUES, C.; FORREST, S. Automatically Finding Patches Using Genetic Programming. In: *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. Washington, DC, USA: IEEE Computer Society, 2009. p. 364–374. Available at: <https://doi.org/10.1109/ICSE.2009.5070536>.

WEN, M.; CHEN, J.; WU, R.; HAO, D.; CHEUNG, S.-C. Context-Aware Patch Generation for Better Automated Program Repair. In: *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. New York, NY, USA: ACM, 2018. p. 1–11. Available at: <https://doi.org/10.1145/3180155.3180233>.

WHITE, M.; TUFANO, M.; MARTINEZ, M.; MONPERRUS, M.; POSHYVANYK, D. Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities. In: *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '19)*. Hangzhou, China: IEEE, 2019. p. 479–490. Available at: <https://doi.org/10.1109/SANER.2019.8668043>.

XIN, Q.; REISS, S. P. Identifying test-suite-overfitted patches through test case generation. In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '17)*. New York, NY, USA: ACM, 2017. p. 226–236. Available at: <https://doi.org/10.1145/3092703.3092718>.

XIN, Q.; REISS, S. P. Leveraging Syntax-Related Code for Automated Program Repair. In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*. Piscataway, NJ, USA: IEEE Press, 2017. p. 660–670. Available at: <https://doi.org/10.1109/ASE.2017.8115676>.

XIONG, Y.; WANG, J.; YAN, R.; ZHANG, J.; HAN, S.; HUANG, G.; ZHANG, L. Precise Condition Synthesis for Program Repair. In: *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. Piscataway, NJ, USA: IEEE Press, 2017. p. 416–426. Available at: <https://doi.org/10.1109/ICSE.2017.45>.

XUAN, J.; MARTINEZ, M.; DEMARCO, F.; CLÉMENT, M.; LAMELAS, S.; DURIEUX, T.; BERRE, D. L.; MONPERRUS, M. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering*, IEEE, v. 43, n. 1, p. 34–55, abr. 2016. Available at: <https://doi.org/10.1109/TSE.2016.2560811>.

YE, H.; MARTINEZ, M.; DURIEUX, T.; MONPERRUS, M. A Comprehensive Study of Automatic Program Repair on the QuixBugs Benchmark. In: *International Workshop on Intelligent Bug Fixing (IBF '19, co-located with SANER)*. Hangzhou, China: IEEE, 2019. p. 1–10. To appear. Available at: <https://arxiv.org/pdf/1805.03454.pdf>.

YU, Z.; MARTINEZ, M.; DANGLOT, B.; DURIEUX, T.; MONPERRUS, M. Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system. *Empirical Software Engineering*, v. 24, n. 1, p. 33–67, Feb 2019. Available at: <https://doi.org/10.1007/s10664-018-9619-4>.

YUAN, Y.; BANZHAF, W. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Transactions on Software Engineering*, 2018. Available at: <https://doi.org/10.1109/TSE.2018.2874648>.

# Appendix

Appendix A

# Evaluation of the Repair Pattern Detector

This appendix reports on a detailed evaluation we performed on the repair pattern detector contained in ADD. We refer to this detector as PPD (Patch Pattern Detector).

**Method.** Our evaluation consists of running PPD on real patches to measure its ability to detect the 25 repair patterns.

*Subject dataset.* The patches used as input to PPD are from Defects4J (JUST et al., 2014), which consists of 395 patches from six real-world projects. We chose this dataset since it contains real bugs and all its patches have been annotated with repair patterns (SOBREIRA et al., 2018), allowing direct comparison between the results generated by PPD and the previous manual detection.

*Result analysis.* We analyzed the results in two steps. First, we calculated the precision and recall of PPD for each pattern, using the available manual detection (SOBREIRA et al., 2018) as an oracle. We refer to such manual detection as *human detection*, while we refer to the detection produced by PPD as *automatic detection*. Second, we performed a manual analysis of the disagreements between the automatic and human detections. For each pattern, the author of this thesis, plus two collaborators, analyzed all patches where there were disagreements and determined whether PPD actually missed or wrongly detected such a pattern. We annotated the disagreements with one of the five diagnostics presented in the first column of Table 26. Then, we calculated the actual precision and recall for each pattern, using the following formulas: $TP = A + B + DC$, $precision = \frac{TP}{TP+DW}$, $recall = \frac{TP}{TP+HC}$, where $A$ is the number of agreements between PPD and human detection, $B$ is the disagreements when both PPD and human detection may be accepted, $DC$ is the disagreements when PPD detection is correct and $DW$ when PPD detection is wrong, and $HC$ is the disagreements when human detection is correct.

Table 25 – PPD performance.

| Pattern | Variant | Prior Precision (%) | Prior Recall (%) | Post Precision (%) | Post Recall (%) |
|---|---|---|---|---|---|
| Conditional Block | Addition | 74.75 | 93.67 | 99.00 | 98.02 |
| | 〃 with Return Statement | 90.12 | 94.81 | 100.00 | 96.47 |
| | 〃 with Exception Throwing | 93.75 | 90.91 | 96.88 | 91.18 |
| | Removal | 60.71 | 77.27 | 86.67 | 89.66 |
| Expression Fix | Logic Modification | 82.22 | 75.51 | 91.11 | 83.67 |
| | 〃 Expansion | 90.20 | 95.83 | 92.16 | 97.92 |
| | 〃 Reduction | 76.92 | 83.33 | 76.92 | 100.00 |
| | Arithmetic Fix | 69.57 | 50.00 | 91.67 | 64.71 |
| Wraps/Unwraps | Wraps-with if | 74.19 | 95.83 | 83.87 | 96.30 |
| | 〃 if-else | 81.25 | 84.78 | 92.00 | 90.20 |
| | 〃 else | 16.67 | 100.00 | 33.33 | 100.00 |
| | 〃 try-catch | 100.00 | 100.00 | 100.00 | 100.00 |
| | 〃 method | 78.57 | 78.57 | 85.71 | 85.71 |
| | 〃 loop | 40.00 | 100.00 | 60.00 | 100.00 |
| | Unwraps-from if-else | 42.11 | 61.54 | 57.89 | 68.75 |
| | 〃 try-catch | 100.00 | 100.00 | 100.00 | 100.00 |
| | 〃 method | 45.45 | 83.33 | 54.55 | 85.71 |
| Single Line | – | 100.00 | 97.96 | 100.00 | 100.00 |
| Wrong Reference | Variable | 66.67 | 76.19 | 82.35 | 89.36 |
| | Method | 68.42 | 83.87 | 86.84 | 89.19 |
| Missing Null-Check | Positive | 95.45 | 84.00 | 100.00 | 100.00 |
| | Negative | 96.67 | 90.63 | 100.00 | 96.77 |
| Copy/Paste | – | 56.16 | 85.42 | 91.78 | 90.54 |
| Constant Change | – | 77.27 | 89.47 | 90.91 | 90.91 |
| Code Moving | – | 60.00 | 85.71 | 81.82 | 100.00 |
| Overall | | 78.26 | 86.95 | 91.53 | 92.39 |

**Results.** The evaluation results are presented in Table 25. For each pattern, this table shows the precision and recall before (column "prior") and after (column "post") the disagreement analysis.

We observed that PPD has a high overall precision and recall, even when comparing it directly with human detection (see the last row in the table). For the most recurring pattern group, *Conditional Block*, both detections agreed on 194 instances of such patterns (prior). After the disagreement analysis, we found that PPD detected 39 new instances of such a pattern, which increased the precision and recall of PPD (post) to at least 86% and 89%, respectively.

For some less recurring patterns, *Single Line* and *Missing Null-Check*, PPD performed well by detecting 96 and 50 instances of these patterns in agreement with human detection, respectively. In fact, for *Single Line*, the only two instances missed by PPD were not truly instances of such a pattern.

However, we identified some particular patterns for which PPD did not perform well. PPD found 95 instances of the patterns from the group *Wraps/Unwraps* in agreement with human detection. On the disagreement analysis, we identified that PPD detected 7

Table 26 – Overall absolute results on the disagreement analysis and reasons for automatic detection differing from manual detection.

| Diagnostic | # Occurrences | Related Reason |
|---|---|---|
| DW (PPD false positive) | 73 | #1, #7 |
| DC (PPD true positive) | 90 | #1, #4 |
| HW (human detection false positive) | 24 | #5, #6 |
| HC (human detection true positive) | 65 | #2, #7 |
| B (both could be accepted) | 33 | #1, #3 |
| A (agreements) | 666 | |
| TP (correct detection = A + B + DC) | 789 | |

new instances of this group, but also generated 30 false positives. The main responsible for these false positives are the pattern variants involving `if`, `else`, and `method`.

**Discussion.** During the disagreement analysis, we also investigated why PPD failed or differed from human analysis. Table 26 relates the diagnostics with the reasons for the disagreements, which we discuss as follows.

*Reason #1:* Global human vision versus AST-based analysis. The GumTree algorithm identifies implicit structures that are not visible to humans. For instance, in Mockito-18, both automatic and manual detections found the pattern *Conditional Block Addition with Return Statement*. However, the automatic detection also found the pattern *Wraps-with if-else*. In this patch, the human sees the structure as in Listing 8, while the structure considered by PPD is like in Listing 9. In other words, the new conditional block wraps a part of the code but with an implicit block. For these occurrences, we considered that both automatic and manual detection could be accepted.

```
+ } else if (type == Iterable.class) {

+    return new ArrayList<Object>(0);
  } else if (type == Collection.class) {

  [...]
```

Listing 8 – Human vision.

```
+ } else {
+    if (type == Iterable.class) {
+       return new ArrayList<Object>(0);
     } else {
        if (type == Collection.class) {
  [...]
```

Listing 9 – AST-based analysis.

Still in the discussion of global human vision versus AST-based analysis, due to fine-grained changes, PPD takes into account small changes that do not make sense as the composition of a pattern in some cases. For instance, PPD detected the *Copy/Paste* pattern in Chart-3. Even though the two additions have high similarity, these changes are not enough to be considered as an instance of the pattern *Copy/Paste*, so we determined this as a false positive generated by PPD.

In the same direction, PPD takes into account *relevant* small changes that humans may not identify in large patches. In these large patches, humans may intuitively consider

only the global vision of the patch and miss smaller changes. For instance, Math-64 has several changes. One of them is the addition of a block with three lines of code at two different locations (i.e., *Copy/Paste*), which was missed by human detection.

*Reason #2:* The automatic detection relies on rules defined by humans (i.e., the author of this thesis and collaborators), and it is difficult to identify all cases where an instance of a pattern may exist, thus PPD missed some pattern instances. The *Expression Fix* detector is the primary responsible for these missed detections. For instance, it missed the detection of an arithmetic expression fix in Math-77, where an arithmetic operation occurs with the assignment operator `+=`, which was replaced by a non-arithmetic assignment operator.

*Reason #3:* There are some borderline cases where a given pattern may fit or not. For instance, on line 1167 of Time-17, one could consider the removed method call as a part of the arithmetic expression used as an argument for such method call, and another one could not. Only manual detection detects such a statement as an instance of the *Arithmetic Expression Fix* pattern, but we considered that detecting it or not can be both accepted.

*Reason #4:* The automatic detection applies the same rules for all patches while it is a difficult task to be done by humans. Therefore, some pattern instances were missed by manual detection due to inconsistencies between patches.

*Reason #5:* In the manual analysis, humans may consider the semantics of changes (even without noticing it) and make assumptions on how the developer could write a patch that matches one of the patterns. For instance, Mockito-28 had been considered as an instance of the *Single Line* pattern. Semantically, it could be correct, but such a pattern should be limited to changes affecting a single line or a single statement in a given patch.

*Reason #6:* A misconception of the patch can impact human analysis. For instance, in Lang-50, manual detection found an instance of the pattern *Logic Expression Modification* on line 285 (and also on the new line 463, which is the same case, i.e., *Copy/Paste*). However, the existing conditional block on line 285 was actually completely changed. The statement inside it was unwrapped in the patch, and the conditional was deleted. Then, an existing conditional block in the code took the place of the conditional considered as having its logic expression modified, i.e., a moving happened, not characterizing a genuine logic expression modification.

*Reason #7:* GumTree is a sophisticated algorithm that may return imprecise results for some patches. For example, it can consider the change over some elements when it is not the case. As a consequence, PPD incorrectly detects or misses some pattern instances.

Appendix B

# Other Bibliographical Contributions

During the three first years of the course of this Ph.D., I worked on a different Software Engineering topic, which resulted in the following publications:

- Fernanda Madeiral, Klérisson V. R. Paixão, Damien Cassou, and Marcelo de Almeida Maia. *Redocumenting APIs with crowd knowledge: a coverage analysis based on question types*. Journal of the Brazilian Computer Society (JBCS), v. 22, n. 9, p. 1–34, 2016.

- Fernanda Madeiral, Klérisson V. R. Paixão, and Marcelo de Almeida Maia. *Redocumentando APIs com Conhecimento da Multidão: um estudo de cobertura da API Swing no Stack Overflow*. Anais do III Workshop Visualização, Evolução e Manutenção de Software (VEM '15), pages 1–8. (Honorable Mention)

I also had the opportunity to collaborate with other researchers, which resulted in the following publication:

- Klérisson V. R. Paixão, Crícia Z. Felício, Fernanda Madeiral, and Marcelo de Almeida Maia. *On the Interplay between Non-Functional Requirements and Builds on Continuous Integration*. Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17)—Mining Challenge Track, pages 479–482.