Automatic Unit Test Amplification for DevOps

PHD THESIS

to obtain the title of

PhD of Science

Specialty : COMPUTER SCIENCE

Expected to be defended on November, 14^{th}, 2019.

Benjamin DANGLOT

prepared at Inria Lille-Nord Europe, SPIRALS Team

Thesis committee:

Supervisors:	Martin MONPERRUS	-	KTH Royal Institute of Technology
	Lionel SEINTURIER	-	Univeristy of Lille
Reviewers:	Lydie DU BOUSQUET	-	University of Joseph Fourier
	Jordi CABOT	-	University of Catalonia
Examiner:	Benoit BAUDRY	-	KTH Royal Institute of Technology
Chair:	Daniel LE BERRE	-	University of Artois
Invited:	Vincent MASSOL	-	XWiki

"O sed fugit interea fugit irreparabile tempus audeamus nunc."

– Rilés

TODO

Acknowledgements

write it

Abstract

Over the last decade, strong unit testing has become an essential component of any serious software project, whether in industry or academia. The agile development movement has contributed to this cultural change with the global dissemination of test-driven development techniques. More recently, the DevOps movement has further strengthened the testing practice with an emphasis on continuous and automated testing. However, testing is tedious and costly for industry: it is hard to evaluate return on investment. Thus, developers under pressure or by lack of discipline or time might skip writing the tests.

To overcome this problem, research investigates the automation of creating strong tests. The dream was that a command-line would give you a complete test suite, that verifies the whole program. Even if automatically generated test suites achieve high coverage, there are still obstacles on the adoption of such techniques by the industry. This can be explained by the difficulties to understand, integrate and maintain generated test suite. Also, most of the tools rely on weak or partial oracles, *e.g.* absence of run-time errors, which limits their ability to find bugs. In this thesis, I aim at addressing the lack of a tool that assists developers in regression testing. To do so, I use test suite amplification.

I defined test amplification and gathered together research works that are using test amplification. I revealed main challenges of test amplification and the main lacks of the state of the art. Test suite amplification consists of exploiting the existing manually-written tests, treated as a seed, to achieve a given engineering goal.

I propose a new approach based on both test input transformation and assertions generation to amplify the test suite. This algorithm is implemented by a tool called DSpot. DSpot works at both level of test methods: input and oracles.

In this thesis, I evaluated DSpot on open-source projects from GitHub. First, I improved the mutation score of test suites and propose these improvement to developers through pull-request. My evaluation showed that developers value the output of DSpot and thus accepted to integrate amplified test methods into their test suite. This evidenced that DSpot can improve the quality of real projects' test suites.

Second, I used DSpot to detect the behavioral difference between two version of the same program. In particular, I used DSpot to detect the behavioral change introduced by a commit in projects on GitHub. This showed that DSpot can be used in the continuous integration to achieve 2 crucial tasks: 1) generate amplified test methods that specify a behavioral change, evolving the test suite; 2) generate amplified test methods to improve the ability to detect potential regressions.

In this thesis, I also expose 3 transversal contributions that are related to the correctness of program. First, the study of programs' correctness under runtime perturbations. Second, the study of pseudo-tested methods that are methods that reveal weaknesses of the test suite. Third, the study of patch overfitting and test generation for automatic repair.

Test amplification

Résumé

write it

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Contents

Nowadays software is omnipresent in people's life: banking, e-commerce, communication, etc. Critical aspects such as aeronautics, voting system or even health care rely on software.

In one of his famous lectures [Dijkstra 1989], Dijkstra has stated that in software:

> *"the smallest possible perturbations - i.e. changes of a single bit - can have the most drastic consequences."*.

Dijkstra highlights the fact that a small fault in software might affect lives. For example, a flight crash happened in 1993 due to an error in the flight-control software of the Swedish JAS 39 Gripen fighter aircraft[1].

To avoid such situations, software engineers adopted testing philosophies: model-checking [Abdulla 2004], proof and code verification [D'Silva 2008], automatic testing. The two former are out of the scope of this thesis, I focus on the latter: the tester writes code to verify that the program is doing what the developer expects. Over the last decade, strong unit testing has become an essential component of any serious software project, whether in industry or academia [Beller 2019, Beller 2015a, Beller 2015b]. The agile development movement has contributed to this cultural change with the global dissemination of test-driven development techniques [Beck 2003]. More recently, the DevOps movement has further strengthened the testing practice with an emphasis on continuous and automated testing [Roche 2013].

[1] see: https://www.flightglobal.com/FlightPDFArchive/1989/1989%20-%200734.PDF

However, testing is tedious and costly for industry: it is hard to evaluate return on investment. Thus, developers under pressure or by lack of discipline or time might skip writing the tests.

To overcome this problem, research investigates the automation of creating strong tests. Automatic generation of tests has been well studied in the last past years [Fraser 2011a, Pacheco 2005a]. The dream was that a command-line would give you a complete test suite, that verifies the whole program.

Researchers investigated the usage of such generated test suite by the industry [Fraser 2015]. Even if automatically generated test suites achieve high coverage, generated test suites do not seem to really help developers. The difficulties to understand generated test suites [Fraser 2015] can be the obstacles on the adoption of such techniques by the industry. Also, most of the tools rely on weak or partial oracles, *e.g.* absence of run-time errors, which limits their ability to find bugs.

In this thesis, I aim at addressing the lack of a tool that assists developers in regression testing. More precisely, the ultimate goal of this thesis is to provide a tool that supports developers in terms of maintenance of their test suite.

To do so, I use test suite amplification. Test suite amplification consists of exploiting the existing manually-written tests, treated as a seed, to achieve a given engineering goal. This seed test method can be seed as a starting point for the test amplification. Test amplification would use all the information carried by this seed test method to perform a given task. For example, a test amplification approach would create variants of this seed test method in order to execute new path in the program, that is to say increasing the code coverage of the test suite. Test suite amplification includes analysis and operation on existing test suites.

In this thesis, I define precisely test suite amplification to leverage this lexical ambiguity and I devise a technique that performs code insertions, deletions or modifications on the seed test method. By construction, test suite amplification's output is close to the test methods used as seed since they share a common-part. It means that for developers that know very-well their software and their test suite, it is easier to understand amplified test methods than test methods that have been generated from scratch without any connection with the human-test methods.

For example, in the context of collaborative project such as projects on GitHub, one developer fixes a bug. If the developer does not provide a test method that exposes the bug fixing, *i.e.* that fails on the version of the program before the fix but passes on the fixed program, the patch might be removed without noticing it. Every changes should come with a test method that characterizes and specifies the changes. For this second objective, the tool would improve the test suite "online", *i.e.* inside the continuous integration service. Each time a developer makes changes, the tool would provide automatically a test method that specifies these changes inside the continuous integration, without any human intervention but the validation of the amplified test method.

The remaining of this chapter is as follow:

I expose the global vision of this thesis in Section 1.1 and its outline in Section 1.2; Then in Section 1.3 present the context of the thesis, the H2020 European project *STAMP*; Eventually, I list the resulting software of this thesis in Section 1.5 and my publications in Section 1.4.

1.1 Scientific Problem

Due the to the democratization of DevOps and agile development methodology, developing teams are committing changes in parallel on the same code base. This way of development brings the advantage to reach the market earlier and adapt the product quickly. However, it has also a major weakness: each change is a potential threat on the correctness of the program. When a developer modify the program code, he might also introduce a bug and/or break an existing feature, *i.e.* add a regression. This can prevent the adoption of the product by customer if clients encounter the bug before any developers. Continuous Integration (CI) mitigates these threats by testing the application often, *e.g.* at each changes, in order to prevent errors in the final product. But the effectiveness of the CI relies on effectiveness of others components such as test suite. If the test suite is poor quality, *e.g.* it covers a small part of the program, the CI might not be able to catch errors. Since developers may cut corners when writing test, **would we be able to provide them a tool to carry automatically the important responsibility that is testing the application?**

First, would this tool be able to improve the test suite in an offline fashion? That is to say, during development, the engineer could use this tool in order to increase the overall quality of the test suite with respect to a given test-criterion, such as mutation score. The mutation score emulates faults that a developer is susceptible to do. In essence, it injects small and artificial behavioral changes that aims at measuring the test suite's capacity to detect them. The challenge is to detect mutants that test methods written by humans do not kill. Moreover, this must be done on an arbitrary programs, using as seed arbitrary test methods. Potentially, the remainder undetected mutants are the most difficult to detect.

Second, could this tool be a part of the continuous integration? That is to say, the tool would automatically run each time that the CI is triggered in order to increase the overall quality of the test suite with respect to the change. In this scenario, the behavioral changes is real, complex and larger than a mutant in mutation score parlance. The challenge is to detect a complex behavioral change the might required a specific knowledge of the program. Thus, the new behavior might require to set the program into a very specific and unknown state, that even the developer is not aware of. In the context of regression testing, the behavioral change, *i.e.* the hidden regression is unknown and might be very difficult to reach. In this thesis, I aim to compete these challenges.

1.2 Thesis Contributions

This thesis makes 5 contributions:

- The definition of "test amplification" and a survey of the literature in Chapter 2. the challenge is that in the litterature, "test amplification" is referred by a various lexicon which makes difficult to understand all the works that use test amplification. Note that this chapter has been published as journal paper [Danglot 2019b].

- The definition of the test suite amplification algorithm and its implementation called DSpot in Chapter 3. The challenge is to design a new algorithm that explores the very large space over the variations of the original methods and to build an oracle. Over the three years of the thesis I have built a tool that addresses these challenges. The tool has been constantly tested and evaluated by industry partners, which are respectable and big companies. This continuous quality feedback has ensured the development of a sound, efficient and useful tool for both researchers and developers.

- The evaluation of DSpot's ability to improve the quality of test suites in Chapter 4. The challenge is to produce amplified test methods that improve the quality of test suites. To do this, I used as study subjects open-source projects that have high quality test suites.

- The evaluation of DSpot's ability to generate amplified tests inside CI in Chapter 5. The challenge is to produce amplified test methods that detect a behavioral difference between two versions of the same programs. For this, I used real commits that change the program's behavior from open-source projects.

In Chapter 6, I expose 3 side contributions made during the thesis. These contributions have been done thanks to the expertise that I acquired during my thesis. However, the reader can skip this chapter because it is additional materials.

1.3 STAMP-project

My thesis takes place within the STAMP project, which is has been funded from the European Union's H2020 research and innovation programme under the grant agreement 731529.

STAMP stands for **S**oftware **T**esting **AMP**lification. STAMP leverages advanced research in test generation and innovative methods of test amplification to push automation in DevOps one step further. The main goal of STAMP techniques are to reduce the number and cost of regression bugs at unit level, configuration level and production stage, by acting at all level of the development cycle.

The project gathers four academic partners with strong software testing expertise, five software companies (in: e-Health, Content Management, Smart Cities and Public Administration), and an open source consortium.

This industry-near research addresses concrete, business-oriented objectives. Thank to STAMP, I could met developers from the industry which are experts in their domain. Thus, It gave me access to case studies that are representative of industrial software.

1.4 Publications

In this section, I list my publications. There are 2 lists: my main contributions and the transversal contributions in which I participated.

These publications have been published in 2 journals:

- Empirical Software Engineering is a journal for applied software engineering research articles with a strong empirical evaluation. Empirical studies in this journal involve large amount of data and an analysis that can be used to evaluate new software practices and technologies.

- Journal of Systems and Software publishes papers on all software engineering and hardware-software-systems issues. These articles must show evidence of their claims, with validations like empirical studies, simulation, etc.

Main contributions:

[Danglot 2019b] Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus and Benoit Baudry. *A Snowballing Literature Study on Test Amplification*. Journal of Systems and Software, page 110398, 2019. (Cited on pages 4, 8 and 35.)

[Danglot 2019] Benjamin Danglot, Oscar Luis Vera-Pérez, Benoit Baudry and Martin Monperrus. *Automatic test improvement with DSpot: a study with ten mature open-source projects*. Empirical Software Engineering, Apr 2019. (Cited on pages 48 and 77.)

[Danglot 2019a] Benjamin Danglot, Martin Monperrus, Walter Rudametkin and Benoit Baudry. *An Approach and Benchmark to Detect Behavioral Changes of Commits in Continuous Integration*. CoRR, vol. abs/1902.08482, 2019. (Cited on page 72.)

Transversal contributions:

[Danglot 2018] Benjamin Danglot, Philippe Preux, Benoit Baudry and Martin Monperrus. *Correctness attraction: a study of stability of software behavior under runtime perturbation*. Empirical Software Engineering, vol. 23, no. 4, pages 2086–2119, Aug 2018. (Cited on pages 99, 102 and 105.)

[Vera-Pérez 2018] Oscar Luis Vera-Pérez, Benjamin Danglot, Martin Monperrus and Benoit Baudry. *A comprehensive study of pseudo-tested methods*. Empirical Software Engineering, Sep 2018. (Cited on pages 77, 99 and 102.)

[Yu 2019] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux and Martin Monperrus. *Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the Nopol repair system*. Empirical Software Engineering, vol. 24, no. 1, pages 33–67, Feb 2019. (Cited on page 102.)

1.5 Software Developed During This Thesis

During my thesis, I developed artifacts to evaluate the approaches. These artifacts are strong enough to be applicable on real code base such open-source projects from GitHub or the STAMP partners' codebases. This shows strong evidence on the applicability and generalization of the results. Following, the list of these artifacts and a small description, all of them are open-source and available on GitHub.

DSpot is a test suite amplifier. It takes as input a project and its test suite and will produce test cases according to a test criterion adequacy such as branch coverage or mutation score. The algorithm implemented by this software is detailed in Chapter 3. URL: `https://github.com/STAMP-project/dspot.git`.

DSpot-diff-test-selection is a maven plugin, based on OpenClover, that produces the list of test classes and their test methods that execute a provided diff. This software allows us to integrate DSpot in the CI. Its evaluation is exposed in Chapter 5. URL: `https://github.com/STAMP-project/dspot/tree/master/dspot-diff-test-selection`.

Test-runner is a library that allows developer to execute test in a new and clean JVM. From my experience, most of research prototypes reinvent the wheel by implementing such library in their own code base. This library provides a generic API to execute tests without any conflict of dependencies, allowing to research prototype to be executed on real programs. This library provides also a way to compute the instruction coverage by the test suite. URL: `https://github.com/STAMP-project/test-runner`.

CHAPTER 2

State of the Art

Contents

Software testing is the art of evaluating an attribute or capability of a program to determine that it meets its required results [Hetzel 1988].

With the advent of agile development methodologies, which advocate testing early and often, a growing number of software projects develop and maintain a test suite [Madeyski 2010]. Those test suites are often large and have been written thanks to a lot of human intelligence and domain knowledge [Zaidman 2011, Zaidman 2008]. Developers spend a lot of time in writing the tests [Beller 2019, Beller 2015a, Beller 2015b], so that those tests exercise interesting cases (including corner cases), and so that an oracle verifies as much as possible the program behavior [Hilton 2018a].

The wide presence of valuable manually written tests has triggered a new thread of research that consists of leveraging the value of existing manually-written tests to achieve a specific engineering goal. This has been coined "test amplification". The term *amplification* is introduced as an umbrella for the various activities that analyze and operate on existing test suites and that are referred to as augmentation, optimization, enrichment, or refactoring in the literature.

This chapter makes the following contributions:

- The first ever snowballing literature review on test amplification
- The classification of the related work into four main categories to help newcomers in the field (students, industry practitioners) understand this body of work.
- A discussion about the outstanding research challenges of test amplification.

Note that this chapter has been published as journal paper [Danglot 2019b] and the remainder is as follows: First, Section 2.1 exposes the methodology used to build this state of the art. This chapter is structured according to the 4 main categories, each of them being presented in a dedicated section. Section 2.2 presents techniques that synthesize new tests from manually-written tests. Section 2.3 focuses on the works that synthesize new tests dedicated to a specific change in the application code (in particular a specific commit). Section 2.4 discusses the less-researched, yet powerful idea of modifying the execution of manually-written tests. Section 2.5 is about the modification of existing tests to improve a specific property. Section 2.6 sums up the analysis. Eventually, Section 2.7 concludes this chapter.

2.1 Method

2.1.1 Definition

Test amplification is defined as follow:

Definition: Test amplification consists of exploiting the knowledge of a large number of test methods, in which developers embed meaningful input data and expected properties in the form of oracles, in order to enhance these manually written tests with respect to an engineering goal (*e.g.* improve coverage of changes or increase the accuracy of fault localization).

Example: A form of test amplification is the addition of test methods automatically generated from the existing manual test methods to increase the coverage of a test suite over the main source code.

Relation to related work: Test amplification is complementary, yet, significantly different from most works on test generation. The key difference is what is given as input to the system. Most test generation tools take as input: the program under test or a formal specification of the testing property. **In contrast, test amplification is defined as taking as primary input test cases written by developers**.

2.1.2 Methodology

Literature studies typically rigorously follow a methodology to ensure both completeness and replication. Cooper's book is taken as reference for a general methodological discussion on literature studies [Cooper 1998]. Specifically for the field of software engineering, well-known methodologies are systematic literature reviews (SLR) [Kitchenham 2004], systematic mapping studies (SMS) [Petersen 2008] and snowballing studies [Wohlin 2014]. For the specific area of *test amplification*, there is no consensus on the terminology used in literature. This is an obstacle to using the SLR and SMS methodologies, which both heavily rely on searching [Brereton 2007]. As snowballing studies are less subject to suffering from the use of diverse terminologies, this study is performed per Wohlin's guidelines [Wohlin 2014, Jalali 2012].

First, I run the search engine of DBLP for all papers containing "test" and "amplification" in their title (using stemming, which means that "amplifying" is matched as well). This has resulted in 70 papers at the date of the search (March 27, 2018)[1]. Each of papers has been reviewed one by one to see whether they fit in the scope according to the definition of subsection 2.1.1. This has resulted in 4 articles [Hamlet 1993, Zhang 2012, Leung 2012, Joshi 2007], which are the seed papers of this literature study. The reason behind this very low proportion (4/70) is that most articles in this DBLP search are in the hardware research community, and hence do not fall in the scope.

Following a breve description of these 4 seed papers:

[1]the data is available at https://github.com/STAMP-project/docs-forum/blob/master/scientific-data/

- [Hamlet 1993] Hamlet and Voas introduce study how different testing planning strategies can amplify testability properties of a software system.

- [Zhang 2012] Zhang and Elbaum explore a new technique to amplify a test suite for finding bugs in exception handling code. Amplification consists in triggering unexpected exceptions in sequences of API calls.

- [Leung 2012] Leung et al propose to modify the test execution by using information gathered from a first test execution. The information is used to derive a formal model used to detect data races in later executions.

- [Joshi 2007] Joshi et al try to amplify the effectiveness of testing by executing both concretely and symbolically the tests.

More details are given in the following sections.

From the seed papers, a backward snowballing search step [Jalali 2012] has been performed, *i.e.*, I have looked at all their references, going backward in the citation graph. 2 of the authors have reviewed the papers, independently. Then, a forward literature search step has been performed, using the Google scholar search engine and "cited by" filter, from the set of papers, in order to find the most recent contributions in this area. A backward snowballing search step and a forward snowballing search step constitute what is called an "iteration". With each iteration, a set of papers is selected for the study, obtained through the snowballing action. These iterations continue until this set of selected paper is empty, *i.e.*, when no paper can be kept, the snowballing process is stopped in both ways: backward and forward.

Once the papers selection is done, 4 key approaches to amplification has been distinguish, which used to classify the literature: Amplification by Adding New Tests as Variants of Existing Ones (Section 2.2); Amplification by Modifying Test Execution (Section 2.3); Amplification by Synthesizing New Tests with Respect to Changes (Section 2.4); Amplification by Modifying Existing Test Code (Section 2.5). The missing terminological consensus mentioned previously prevented the design of a classification according to Petersen's guidelines [Petersen 2008]. The four categories has been incrementally refined by analyzing the techniques and goals in each paper. The methodology is as follows: a work is assigned to a category if the key technique of the paper corresponds to it. If no category captures the gist of the paper, a new category is created. Two categories that are found to be closely related are merged to create a new one. The incremental refinement of these findings led to the definition of 4 categories to organize this literature study.

2.2 Amplification by Adding New Tests as Variants of Existing Ones

The most intuitive form of test amplification is to consider an existing test suite, then generate variants of the existing test cases and add those new variants into the original test suite. This kind of test amplification is denoted as AMP_{add}.

> **Definition:** A test amplification technique AMP_{add} consists of creating new tests from existing ones to achieve a given engineering goal. The most commonly used engineering goal is to improve coverage according to a coverage criterion.

The works listed in this section fall into this category and have been divided according to their main engineering goal.

2.2.1 Coverage or Mutation Score Improvement

Baudry *et al.* [Baudry 2005b] [Baudry 2005a] improve the mutation score of an existing test suite by generating variants of existing tests through the application of specific transformations of the test cases. They iteratively run these transformations, and propose an adaptation of genetic algorithms (GA), called a bacteriological algorithm (BA), to guide the search for test cases that kill more mutants. The results demonstrate the ability of search-based amplification to significantly increase the mutation score of a test suite. They evaluated their approach on 2 case studies that are .NET classes. The evaluation shows promising results, however the result have little external validity since only 2 classes are considered.

Tillmann and Schulte [Tillmann 2006] describe a technique that can generalize existing unit tests into parameterized unit tests. The basic idea behind this technique is to refactor the unit test by replacing the concrete values that appear in the body of the test with parameters, which is achieved through symbolic execution. Their technique's evaluation has been conducted on 5 .NET classes.

The problem of generalizing unit tests into parameterized unit tests is also studied by Thummalapenta et *et al.* [Marri 2010]. Their empirical study shows that unit test generalization can be achieved with feasible effort, and can bring the benefits of additional code coverage. They evaluated their approach on 3 applications from 1 600 to 6 200 lines of code. The result shows an increase of the branch coverage and a slight increase of the bug detection capability of the test suite.

To improve the cost efficiency of the test generation process, Yoo and Harman [Yoo 2012] propose a technique for augmenting the input space coverage of the existing tests with new tests. The technique is based on four transformations on numerical values in test cases, *i.e.* shifting ($\lambda x.x + 1$ and $\lambda x.x - 1$) and data scaling (multiply or divide the value by 2). In addition, they employ a hill-climbing algorithm based on the number of fitness function evaluations, where a fitness is the computation of the euclidean distance between two input points in a numerical space. The empirical evaluation shows that the technique can achieve better coverage than some test generation methods which generate tests from scratch. The approach has been evaluated on the triangle problem. They also evaluated their approach on two specific methods from two large and complex libraries.

To maximize code coverage, Bloem *et al.* [Bloem 2014] propose an approach that alters existing tests to get new tests that enter new terrain, *i.e.* uncovered features of the program. The approach first analyzes the coverage of existing tests, and then selects all test cases that pass a yet uncovered branch in the target function. Finally, the approach investigates the path conditions of the selected test cases one by one to get a new test that covers a previously uncovered branch. To vary path conditions of existing tests, the approach uses symbolic execution and model checking techniques. A case study has shown that the approach can achieve 100% branch coverage fully automatically. They first evaluate their prototype implementation on two open source examples and then present a case study on a real industrial program of a Java Card applet firewall. For the real program, they applied their tool on 211 test cases, and produce 37 test cases to increase the code coverage. The diversity of the benchmark allows to make a first generalization.

Rojas et al. [Rojas 2016] have investigated several seeding strategies for the test generation tool Evosuite. Traditionally, Evosuite generates unit test cases from scratch. In this context, seeding consists in feeding Evosuite with initial material from which the automatic generation process can start. The authors evaluate different sources for the seed: constants in the program, dynamic values, concrete types and existing test cases. In the latter case, seeding analogizes to amplification. The experiments with 28 projects from the Apache Commons repository show a 2% improvement of code coverage, on average, compared to a generation from scratch. The evaluation based on Apache artifacts is stronger than most related work, because Apache artifacts are known to be complex and well tested.

Patrick and Jia [Patrick 2017] propose *Kernel Density Adaptive Random Testing* (KD-ART) to improve the effectiveness of random testing. This technique takes advantage of run-time test execution information to generate new test inputs. It first applies *Adaptive Random Testing* (ART) to generate diverse values uniformly distributed over the input space. Then, they use *Kernel Density Estimation* for estimating the distribution of values found to be useful; in this case, that increases the mutation score of the test suite. KD-ART can intensify the existing values by generating inputs close to the ones observed to be more useful or diversify the current inputs by using the ART approach. The authors explore the trade-offs between diversification and intensification in a benchmark of eight C programs. They achieve an 8.5% higher mutation score than ART for programs that have simple numeric input parameters, but their approach does not show a significant increase for programs with composite inputs. The technique is able to detect mutants 15.4 times faster than ART in average.

Instead of operating at the granularity of complete test cases, Yoshida et *et al.* [Yoshida 2016] propose a novel technique for automated and fine-grained incremental generation of unit tests through minimal augmentation of an existing test suite. Their tool, *FSX*, treats each part of existing cases, including the test driver, test input data, and oracles, as “test intelligence", and attempts to create tests for uncovered test targets by copying and minimally modifying existing tests wherever possible. To achieve this, the technique uses iterative, incremental refinement of test-drivers and symbolic execution. They evaluated *FSX* using four benchmarks, from 5K to 40K lines of code. This evaluation is adequate and reveals that FSX’ result can be generalized.

2.2.2 Fault Detection Capability Improvement

Starting with the source code of test cases, Harder et *et al.* [Harder 2003] propose an approach that dynamically generates new test cases with good fault detection ability. A generated test case is kept only if it adds new information to the specification. They define “new information” as adding new data for mining invariants with Daikon, hence producing new or modified invariants. What is unique in the paper is the augmentation criterion: helping an invariant inference technique. They evaluated Daikon on a benchmark of 8 C programs. These programs vary from 200 to 10K line of code. It is left to future work to evaluate the approach on a real and large software application.

Pezze et *et al.* [Pezze 2013] observe that method calls are used as the atoms to construct test cases for both unit and integration testing, and that most of the code in integration test cases appears in the same or similar form in unit test cases. Based on this observation, they propose an approach which uses the information provided in unit test cases about object creation and initialization to build composite cases that focus on testing the interactions between objects. The evaluation results show that the approach can reveal new interaction faults even in well tested applications.

Writing web tests manually is time consuming, but it gives the developers the advantage of gaining domain knowledge. In contrast, most web test generation techniques are automated and systematic, but lack the domain knowledge required to be as effective. In light of this, Milani *et al.* [Milani Fard 2014] propose an approach which combines the advantages of the two. The approach first extracts knowledge such as event sequences and assertions from the human-written tests, and then combines the knowledge with the power of automated crawling. It has been shown that the approach can effectively improve the fault detection rate of the original test suite. They conducted an empirical evaluation on 4 open-source and large JavaScript systems.

2.2.3 Oracle Improvement

Pacheco and Ernst implement a tool called Eclat [Pacheco 2005b], which aims to help the tester with the difficult task of creating effective new test inputs with constructed oracles. Eclat first uses the execution of some available correct runs to infer an operational model of the software's operation. By making use of the established operational model, Eclat then employs a classification-guided technique to generate new test inputs. Next, Eclat reduces the number of generated inputs by selecting only those that are most likely to reveal faults. Finally, Eclat adds an oracle for each remaining test input from the operational model automatically. They evaluated their approach on 6 small programs. They compared Eclat's result to the result of JCrasher, a state of the art tool that has the same goal than Eclat. In their experimentation, they report that Eclat perform better than JCrasher: Eclat reveals 1.1 faults on average against 0.02 for JCrasher.

Given that some test generation techniques just generate sequences of method calls but do not contain oracles for these method calls, Fraser and Zeller [Fraser 2011c] propose an approach to generate parametrized unit tests containing symbolic pre- and post-conditions. Taking concrete inputs and results as inputs, the technique uses test generation and mutation to systematically generalize pre- and post-conditions. Evaluation results on five open source libraries show that the approach can successfully generalize a concrete test to a parameterized unit test, which is more general and expressive, needs fewer computation steps, and achieves a higher code coverage than the original concrete test. They used 5 open-source and large programs to evaluate the approach. According to their observation, this technique is more expensive than simply generating unit test cases.

2.2.4 Debugging Effectiveness Improvement

Baudry *et al.* [Baudry 2006] propose the test-for-diagnosis criterion (TfD) to evaluate the fault localization power of a test suite, and identify an attribute called Dynamic Basic Block (DBB) to characterize this criterion. A Dynamic Basic Block (DBB) contains the set of statements that are executed by the same test cases, which implies all statements in the same DBB are indistinguishable. Using an existing test suite as a starting point, they apply a search-based algorithm to optimize the test suite with new tests so that the test-for-diagnosis criterion can be satisfied. They evaluated their approach on two programs: a toy program and a server that simulates business meetings over the network. These two programs are less than 2K line of code long, which can be considered as small.

Röβler *et al.* [Röβler 2012] propose BugEx, which leverages test case generation to systematically isolate failure causes. The approach takes a single failing test as input and starts generating additional passing or failing tests that are similar to the failing test. Then, the approach runs these tests and captures the differences between these runs in terms of the observed facts that are likely related with the pass/fail outcome. Finally, these differences are statistically ranked and a ranked list of facts is produced. In addition, more test cases are further generated to confirm or refute the relevance of a fact. It has been shown that for six out of seven real-life bugs, the approach can accurately pinpoint important failure explaining facts. To evaluate BugEx, they use 7 real-life case studies from 68 to 62K lines of code. The small number of considered bugs, 7, calls for more research to improve external validity.

Yu *et al.* [Yu 2013] aim at enhancing fault localization under the scenario where no appropriate test suite is available to localize the encountered fault. They propose a mutation-oriented test case augmentation technique that is capable of generating test suites with better fault localization capabilities. The technique uses some mutation operators to iteratively mutate some existing failing tests to derive new test cases potentially useful to localize the specific encountered fault. Similarly, to increase the chance of executing the specific path during crash reproduction, Xuan *et al.* [Xuan 2015] propose an approach based on test case mutation. The approach first selects relevant test cases based on the stack trace in the crash, followed by eliminating assertions in the selected test cases, and finally uses a set of predefined mutation operators to produce new test cases that can help to reproduce the crash. They evaluated MuCrash on 12 bugs for Apache Commons Collections, which is 26 KLoC of source code and 29 KLoC of test code length. The used program is quite large and open-source which increases the confidence. but using a single subject is a threat to generalization.

2.2.5 Summary

Main achievements: The works discussed in this section show that adding new test cases based on existing ones can make the test generation process more targeted and cost-effective. On the one hand, the test generation process can be geared towards achieving a specific engineering goal better based on how existing tests perform with respect to the goal. For instance, new tests can be intentionally generated to cover those program elements that are not covered by existing tests. Indeed, it has been shown that tests generated in this way are effective in achieving multiple engineering goals, such as improving code coverage, fault detection ability, and debugging effectiveness. On the other hand, new test cases can be generated more cost-effectively by making use of the structure or components of the existing test cases.

Main Challenges: While existing tests provide a good starting point, there are some difficulties in how to make better use of the information they contain. First, the number of new tests synthesized from existing ones can sometimes be large and hence an effective strategy should be used to select tests that help to achieve the specific engineering goal; the concerned works are: [Baudry 2005b, Baudry 2005a, Yoshida 2016].

Second, the synthesized tests have been applied to a specific set of programs and the generalization of the related approaches could be limited. The concerned works are: [Tillmann 2006, Marri 2010, Yoo 2012, Bloem 2014, Patrick 2017, Harder 2003, Pacheco 2005b, Baudry 2006, Röβler 2012, Xuan 2015]. Third, some techniques have known performance issues and do not scale well: [Milani Fard 2014, Fraser 2011c].

2.3 Amplification by Synthesizing New Tests with Respect to Changes

Software applications are typically not tested at a single point in time; they are rather tested incrementally, along with the natural evolution of the code base: new tests are typically added together with a change or a commit [Zaidman 2011, Zaidman 2008], to verify, for instance, that a bug has been fixed or that a new feature is correctly implemented. In the context of test amplification, it directly translates to the idea of synthesizing new tests as a reaction to a change. This can be seen as a specialized form AMP_{add}, which considers a specific change, in addition to the existing test suite, to guide the amplification. This kind of test amplification is denoted as AMP_{change}.

Definition: Test amplification technique AMP_{change} consists of adding new tests to the current test suite, by creating new tests that cover and/or observe the effects of a change in the application code.

I first present a series of works by Xu *et al.*, who develop and compare two alternatives of test suite augmentation, one based on genetic algorithms and the other on concolic execution. A second subsection presents the work of a group of authors that center the attention on finding testing conditions to exercise the portions of code that exhibit changes. A third subsection exposes works that explore the adaptation and evolution of test cases to cope with code changes. The last subsection shows other promising works in this area.

2.3.1 Search-based vs. Concolic Approaches

In their work, Xu *et al.* [Xu 2009] focus on the scenario where a program has evolved into a new version through code changes in development. They consider techniques as (i) the identification of coverage requirements for this new version, given an existing test suite; and (ii) the creation of new test cases that exercise these requirements. Their approach first identifies the parts of the evolved program that are not covered by the existing test suite. In the same process they gather path conditions for every test case. Then, they exploit these path conditions with a concolic testing method to find new test cases for uncovered branches, analyzing one branch at a time.

Symbolic execution is a program analysis technique to reason about the execution of every path and to build a symbolic expression for each variable. Concolic testing also carries a symbolic state of the program, but overcomes some limitations of a fully symbolic execution by also considering certain concrete values. Both techniques are known to be computationally expensive for large programs.

Xu *et al.* avoid a full concolic execution by only targeting paths related to uncovered branches. This improves the performance of the augmentation process. They applied their technique to 22 versions of a small arithmetic program from the SIR [SIR] repository and achieved branch coverage rates between 95% and 100%. They also show that a full concolic testing is not able to obtain such high coverage rates and needs a significantly higher number of constraint solver calls.

In subsequent work, Xu *et al.* [Xu 2010a] address the same problem with a genetic algorithm. Each time the algorithm runs, it targets a branch of the new program that is not yet covered. The fitness function measures how far a test case falls from the target branch during its execution. The authors investigate if all test cases should be used as population, or only a subset related to the target branch or, if newly generated cases should be combined with existing ones in the population. Several variants are compared according to their efficiency and effectiveness, that is, whether the generated test cases achieve the goal of exercising the uncovered branches. The experimentation targets 3 versions of *Nanoxml*, an XML parser implemented in Java with more than 7 KLoC and included in the SIR [SIR] repository. The authors conclude that considering all tests achieves the best coverage, but also requires more computational effort. They imply that the combination of new and existing test cases is an important factor to consider in practical applications.

Xu *et al.* then dedicate a paper to the comparison of concolic execution and genetic algorithms for test suite amplification [Xu 2010b]. The comparison is carried out over four small (between 138 and 516 LoC) C programs from the SIR [SIR] repository. They conclude that both techniques benefit from reusing existing test cases at a cost in efficiency. The authors also state that the concolic approach can generate test cases effectively in the absence of complex symbolic expressions. Nevertheless, the genetic algorithm is more effective in the general case, but could be more costly in test case generation. Also, the genetic approach is more flexible in terms of scenarios where it can be used, but the quality of the obtained results is heavily influenced by the definition of the fitness function, mutation test and crossover strategy.

The same authors propose a hybrid approach [Xu 2011]. This new approach incrementally runs both the concolic and genetic methods. Each round applies first the concolic testing and the output is passed to the genetic algorithm as initial population. Their original intention was to get a more cost-effective approach. The evaluation is done over three of the C programs from their previous study. The authors conclude that this new proposal outperforms the other two in terms of branch coverage, but in the end is not more efficient. They also speculate about possible strategies for combining both individual approaches to overcome their respective weaknesses and exploit their best features. A revised and extended version of this work is given in [Xu 2015].

2.3.2 Finding Test Conditions in the Presence of Changes

Another group of authors have worked under the premise that achieving only coverage may not be sufficient to adequately exercise changes in code. Sometimes these changes manifest themselves only when particular conditions are met by the input. The following papers address the problem of finding concrete input conditions that not only can execute the changed code, but also propagate the effects of this change to an observable point that could be the output of the involved test cases. However, their work does not create concrete new test cases. Their goal is to provide guidance, in the form of conditions that can be leveraged to create new tests with any generation method.

It is important to notice that they do not achieve test generation. Their goal is to provide guidance to generate new test cases independently of the selected generation method.

Apiwattanapong *et al.* [Apiwattanapong 2006] target the problem of finding test conditions that could propagate the effects of a change in a program to a certain execution point. Their method takes as input two versions of the same program. First, an alignment of the statements in both versions is performed. Then, starting from the originally changed statement and its counterpart in the new version, all statements whose execution is affected by the change are gathered up to a certain distance. The distance is computed over the control and data dependency graph. A partial symbolic execution is performed over the affected instructions to retrieve the states of both program versions, which are in turn used to compute testing requirements that can propagate the effects of the original change to the given distance. As said before, the method does not deal with test case creation, it only finds new testing conditions that could be used in a separate generation process and is not able to handle changes to several statements unless the changed statements are unrelated. The approach is evaluated on Java translations of two small C programs (102 Loc and 268 LoC) originally included in the Siemens program dataset [Hutchins 1994]. The authors conclude that, although limited to one change at a time, the technique can be leveraged to generate new test cases during regular development.

Santelices *et al.* [Santelices 2008] continue and extend the previous work by addressing changes to multiple statements and considering the effects they could have on each other. In order to achieve this they do not compute state requirements for changes affected by others. This time, the evaluation is done in one of the study subjects form their previous study and two versions of *Nanoxml* from SIR.

In another paper [Santelices 2011] the same authors address the problems in terms of efficiency of applying symbolic execution. They state that limiting the analysis of affected statements up to a certain distance from changes reduces the computational cost, but scalability issues still exist. They also explain that their previous approach often produces test conditions which are unfeasible or difficult to satisfy within a reasonable resource budget. To overcome this, they perform a dynamic inspection of the program during test case execution over statically computed slices around changes. The technique is evaluated over five small Java programs, comprising *Nanoxml* with 3 KLoC and translations of C programs from SIR having between 283 LoC and 478 LoC. This approach also considers multiple program changes. Removing the need of symbolic execution leads to a less expensive method. The authors claim that propagation-based testing strategies are superior to coverage-based in the presence of evolving software.

2.3.3 Other Approaches

Other authors have also explored test suite augmentation for evolving programs with propagation-based approaches. Qui *et al.* [Qi 2010] propose a method to add new test cases to an existing test suite ensuring that the effects of changes in the new program version are observed in the test output. The technique consists of a two step symbolic execution. First, they explore the paths towards a change in the program guided by a notion of distance over the control dependency graph. This exploration produces an input able to reach the change. In a second moment they analyze the conditions under which this input may affect the output and make changes to the input accordingly. The technique is evaluated using 41 versions of the *tcas* program from the SIR repository (179 LoC) with only one change between versions. The approach was able to generate tests reaching the changes and affected the program output for 39 of the cases. Another evaluation was also included for two consecutive versions of the *libPNG* library (28 KLoC) with a total of 10 independent changes between them. The proposed technique was able to generate tests that reached the changes in all cases and the output was affected in nine of the changes. The authors conclude that the technique is effective in the generation of test inputs to reach a change in the code and expose the change in the program output.

Wang *et al.* [Wang 2014] exploit existing test cases to generate new ones that execute the change in the program. These new test cases should produce a new program state, in terms of variable values, that can be propagated to the test output. An existing test case is analyzed to check if it can reach the change in an evolved program. The test is also checked to see if it produces a different program state at some point and if the test output is affected by the change. If some of these premises do not hold then the path condition of the test is used to generate a new path condition to achieve the three goals. Further path exploration is guided and narrowed using a notion of the probability for the path condition to reach the change. This probability is computed using the distance between statements over the control dependency graph. Practical results of test cases generation in three small Java programs (from 231 LoC to 375 LoC) are exhibited. The method is compared to *eXpress* and *JPF-SE* two state of the art tools and is shown to reduce the number of symbolic executions by 45.6% and 60.1% respectively. As drawback, the technique is not able to deal with changes on more than one statement.

Mirzaaghaei *et al.* [Mirzaaghaei 2012, Mirzaaghaei 2014] introduce an approach that leverages information from existing test cases and automatically adapts test suites to code changes. Their technique can repair, or evolve test cases in front of signature changes (*i.e.* changing the declaration of method parameters or return values), the addition of new classes to the hierarchy, addition of new interface implementations, new method overloads and new method overrides. Their effective implementation *TestCareAssitance* (TCA) first diffs the original program with its modified version to detect changes and searches in the test code similar patterns that could be used to complete the missing information or change the existing code. They evaluate TCA for signature changes in 9 Java projects of the Apache foundation and repair in average 45% of modifications that lead to compilation errors. The authors further use five additional open source projects to evaluate their approach when adding new classes to the hierarchy. TCA is able to generate test cases for 60% of the newly added classes. This proposal could also fall in the category of test repairing techniques. Section 2.4 will explore alternatives in a similar direction that produce test changes instead of creating completely new test cases.

In a different direction, Böhme *et al.* [Böhme 2013] explain that changes in a program should not be treated in isolation. Their proposal focuses on potential interaction errors between software changes. They propose to build a graph containing the relationship between changed statements in two different versions of a program and potential interaction locations according to data and control dependency. This graph is used to guide a symbolic execution method and find path conditions for exercising changes and their potential interactions and use a Satisfiability Modulo Solver to generate a concrete test input. They provide practical results on six versions the *GNU Coreutils* toolset that introduce 11 known errors. They were able to find 5 unknown errors in addition to previously reported issues.

Marinescu and Cadar [Marinescu 2013] present a system, called *Katch*, that aims at covering the code included in a patch. Instead of dealing with one change to one statement, as most of the previous works, this approach first determines the differences of a program and its previous version after a commit, in the form of a code patch. Lines included in the patch are filtered by removing those that contain non-executable code (*i.e.* comments, declarations). If several lines belong to the same basic program block, only one of them is kept as they will all be executed together. From the filtered set of lines, those not covered by the existing test suite are considered as targets. The approach then selects the closest input to each target from existing tests using the static minimum distance over the control flow

graph. Edges on this graph that render the target unreachable are removed by inspecting the data flow and gathering preconditions to the execution of basic blocks. To generate new test inputs, they combine symbolic execution with heuristics that select branches by their distance to the target, regenerate a path by going back to the point where the condition became unfeasible or changing the definition of variables involved in the condition. The proposal is evaluated using the *GNU findutils*, *diffutils* and *binutils* which are distributed with most Unix-based distributions.
They examine patches from a period of 3 years. In average, they automatically increase coverage from 35% to 52% with respect to the manually written test suite.

A posterior work of the same group [Palikareva 2016] also targets patches of code, focusing on finding test inputs that execute different behavior between two program versions. They consider two versions of the same program, or the old version with the patch of changed code, and a test suite. The code should be annotated in places where changes occur in order to unify both versions of the program for the next steps. Then they select from the test suite those test cases that cover the changed code. If there is no such test case, it can be generated using *Katch*. The unified program is used in a two stage dynamic symbolic execution guided by the selected test cases: look for branch points where two semantically different conditions are evaluated in both program versions; bounded symbolic execution for each point previously detected. At those points all possible alternatives in which program versions execute the same or different branch blocks are considered and used to make the constraint solver generate new test inputs for divergent scenarios. The program versions are then normally executed with the generated inputs and the result is validated to check the presence of a bug or an intended difference. In their experiments this validation is mostly automatic but in general should be performed by developers. The evaluation of the proposed method is based on the *CoREBench* [Böhme 2014] data set that contains documented bugs and patches of the *GNU Coreutils* program suite. The authors discuss successful and unsuccessful results but in general the tool is able to produce test inputs that reveal changes in program behaviour.

2.3.4 Summary

Main achievements: AMP_{change} techniques often rely on symbolic and concolic execution. Both have been successfully combined with other techniques in order to generate test cases that reach changed or evolved parts of a program [Xu 2011, Xu 2015, Marinescu 2013]. Those hybrid approaches produce new test inputs that increase the coverage of the new program version. Data and control dependency has been used in several approaches to guide symbolic execution and reduce its computational cost [Böhme 2013, Marinescu 2013, Wang 2014]. The notion of distance from statements to observed changes has been also used for this matter [Marinescu 2013, Apiwattanapong 2006].

Main challenges: Despite the progress made in the area, a number of challenges remain open. The main challenge relates to the size of the changes considered for test amplification: many of the works in this area consider a single change in a single statement [Apiwattanapong 2006, Qi 2010, Wang 2014]. While this is relevant and important to establish the foundations for AMP_{change}, this cannot fit current development practices where a change, usually a commit, modifies the code at multiple places at once. A few papers have started investigating multi-statement changes for test suite amplification [Santelices 2008, Marinescu 2013, Palikareva 2016]. Now, AMP_{change} techniques should fit into the revision process and be able to consider a commit as the unit of change.

Another challenge relates to scalability. The use of symbolic and concolic execution has proven to be effective in test input generation targeting program changes. Yet, these two techniques are computationally expensive [Xu 2009, Xu 2011, Xu 2015, Apiwattanapong 2006, Santelices 2008, Palikareva 2016]. Future works shall consider more efficient ways for exploring input requirements that exercise program changes or new uncovered parts. Santelices and Harrold [Santelices 2011] propose to get rid of symbolic execution by observing the program behavior during test execution. However, they do not generate test cases.

Practical experimentation and evaluation remains confined to a very small number of programs, in most cases less than five study subjects, and even small programs in terms of effective lines of code. A large scale study on the subject is still missing.

2.4 Amplification by Modifying Test Execution

In order to explore new program states and behavior, it is possible to interfere with the execution at runtime so as to modify the execution of the program under test.

Definition: Test amplification technique AMP_{exec} consists of modifying the test execution process or the test harness in order to maximize the knowledge gained from the testing process.

One of the drawbacks of automated tests is the hidden dependencies that may exist between different unit test cases. In fact, the order in which the test cases are executed may affect the state of the program under test. A good and strong test suite should have no implicit dependencies between test cases.

The majority of test frameworks are deterministic, *i.e.* between two runs the order of execution of test is the same [Palomba 2017, Palomb].

An AMP_{exec} technique would randomize the order in which the tests are executed to reveal hidden dependencies between unit tests and potential bugs derived from this situation.

2.4.1 Amplification by Modifying Test Execution

Zhang and Elbaum [Zhang 2012, Zhang 2014] describe a technique to validate exception handling in programs making use of APIs to access external resources such as databases, GPS or bluetooth. The method mocks the accessed resources and amplifies the test suite by triggering unexpected exceptions in sequences of API calls. Issues are detected during testing by observing abnormal terminations of the program or abnormal execution times. They evaluated their approach on 5 Android artifacts. Their sizes vary from 6k to 18k line of codes, with 39 to 117 unit tests in the test suite. The size of the benchmark seems quite reasonable. The approach is shown to be cost-effective and able to detect real-life problems in 5 Android applications.

Cornu *et al.* [Cornu 2015] work in the same line of exception handling evaluation. They propose a method to complement a test suite in order to check the behaviour of a program in the presence of unanticipated scenarios. The original code of the program is modified with the insertion of `throw` instructions inside `try` blocks. The test suite is considered as an executable specification of the program and therefore used as an oracle in order to compare the program execution before and after the modification. Under certain conditions, issues can be automatically repaired by catch-stretching. The authors used 9 Java open-source projects to create a benchmark and evaluate their approach. This benchmark is big enough to conclude the generalization of the results. The selected artifacts are well-known, modern and large: Apache artifacts, joda-time and so on. Their empirical evaluation shows that the short-circuit testing approach of exception contracts increases the knowledge of software.

Leung *et al.* [Leung 2012] are interested in finding data races and non-determinism in GPU code written in the CUDA programming language. In their context, test amplification consists of generalizing the information learned from a single dynamic run. The main contribution is to formalize the relationship between the trace of the dynamic run and statically collected information flow. The authors leverage this formal model to define the conditions under which they can generalize the absence of race conditions for a set of input values, starting from a run of the program with a single input. They evaluated their approach using 28 benchmarks in the NVIDIA CUDA SDK Version 3.0. They removed trivial ones and some of them that they cannot handle. The set of benchmarks is big enough and contains a diversity of applications to be convinced that the approach can be generalized.

Fang *et al.* [Fang 2015] develop a performance testing system named *Perfblower*, which is able to detect and diagnose memory issues by observing the execution of a set of test methods. The system includes a domain-specific language designed to describe memory usage symptoms. Based on the provided descriptions, the tool evaluates the presence of memory problems. The approach is evaluated on 13 Java real-life projects. The tool is able to find real memory issues and reduce the number of false positives reported by similar tools. They used the small workload of the DaCapo [Blackburn 2006] benchmark. They argue that developers will not use large workloads and it is much more difficult to reveal performance bugs under small workloads. These two claims are legit, however the authors do not provide any evidence of the scalability of the approach.

Zhang *et al.* [Zhang 2016] devise a methodology to improve the capacity of the test suite to detect regression faults. Their approach is able to exercise uncovered branches without generating new test cases. They first look for identical code fragments between a program and its previous version. Then, new variants of both versions are generated by negating branch conditions that force the test suite to execute originally uncovered parts. The behavior of version variants are compared through test outputs. An observed difference in the output could reveal an undetected fault. An implementation of the approach is compared with *EvoSuite* [Fraser 2011b] on 10 real-life Java projects. In the experiments, known faults are seeded by mutating the original program code. The results show that *EvoSuite* obtains better branch coverage, while the proposed method is able to detect more faults. The implementation is available in the form of a tool named *Ison*.

2.4.2 Summary

Main achievements: AMP_{exec} proposals provide cost-effective approaches to observe and modify a program execution to detect possible faults. This is done by instrumenting the original program code to place observations at certain points or mocking resources to monitor API calls and explore unexpected scenarios. It adds no prohibitive overheads to regular test execution and provides means to gather useful runtime information. Techniques in this section were used to analyze real-life projects of different sizes and they are shown to match other tools that pursue the same goal and obtain better results in some cases.

Main challenges: As shown by the relatively small number of papers discussed in this section, techniques for test execution modification have not been widely explored. The main challenge is to get this concept known so as to enlarge the research community working on this topic. The concerned works are: [Zhang 2012, Zhang 2014, Cornu 2015, Leung 2012, Fang 2015, Zhang 2016].

2.5 Amplification by Modifying Existing Test Code

In testing, it is up to the developer to design integration (large) or unit (small) tests. The main testing infrastructure such as JUnit in Java does not impose anything on the tests, such as the number of statements in a test, the cohesion of test assertions or the meaningfulness of test methods grouped in a test class. In literature, there are works on modifying existing tests with respect to a certain engineering goal.

Definition: Test amplification technique AMP_{mod} refers to modifying the body of existing test methods. The goal here is to make the scope of each test methods more precise or to improve the ability of test cases at assessing correctness (with better oracles). Differently from AMP_{add}, it is not about adding new test methods or new tests classes.

2.5.1 Input Space Exploration

Dallmeier *et al.* [Dallmeier 2010] automatically amplify test suites by adding and removing method calls in JUnit test cases. Their objective is to produce test methods that cover a wider set of executions than the original test suite in order to improve the quality of models reverse engineered from the code. They evaluate *TAUTOKO* on 7 Java classes and show that it is able to produce richer typestates (a typestate is a finite state automaton which encodes legal usages of a class under test).

Hamlet and Voas [Hamlet 1993] introduce the notion of "reliability amplification" to establish a better statistical confidence that a given software is correct. Program reliability is measured as the mean time to failure of the system under test. The core contribution relates reliability to testability assessment, that is, a measure of the probability that a fault in the program will propagate to an observable state. The authors discuss how different systematic test planning strategies, *e.g.* partition-based test selection [Ostrand 1988], can complement profile-based test cases, in order to obtain a better measurement of testability and therefore better bounds to estimate the reliability of the program being tested.

2.5.2 Oracle Improvement

Xie [Xie 2006] amplifies object-oriented unit tests with a technique that consists of adding assertions on the state of the receiver object, the returned value by the tested method (if it is a non-void return value method) and the state of parameters (if they are not primitive values). Those values depend on the behavior of the given method, which in turn depends on the state of the receiver and of arguments at the beginning of the invocation. The approach, named *Orstra*, consists of instrumenting the code and running the test suite to collect state of objects. Then, assertions are generated, which call observer methods (methods with a non-void return type, *e.g.* $toString()$). To evaluate *Orstra*, the author uses 11 Java classes from a variety of sources. Theses classes are different in the number of methods and lines of code, and the author also uses two different third-party test generation tools to generate the initial test suite to be amplified. The results show that *Orstra* can effectively improve the fault-detection capability of the original automatically generated test suite.

Carzaniga *et al.* [Carzaniga 2014] reason about generic oracles and propose a generic procedure to assert the behavior of a system under test. To do so, they exploit the redundancy of software. Redundancy of software happens when the system can perform the same action through different executions, either with different code or with the same code but with different input parameters or in different contexts. They devise the notion of "cross-checking oracles", which compare the outcome of the execution of an original method to the outcome of an equivalent method. Such oracle uses a generic equivalence check on the returned values and the state of the target object. If there is an inconsistency, the oracle reports it, otherwise, the checking continue. These oracles are added to an existing test suite with aspect-oriented programming. For the evaluation, they use 18 classes from three non-trivial open-source Java libraries, including Guava, Joda-Time, and GraphStream. The subject classes are selected based on whether a set of equivalences have already been established or could be identified. For each subject class, two kinds of test suites have been used, including hand-written test suites and automatically generated test suites by Randoop. The experimental results show that the approach can slightly increase (+6% overall) the mutation score of a manual test suite.

Joshi *et al.* [Joshi 2007] try to amplify the effectiveness of testing by executing both concretely and symbolically the tests. Along this double execution, for every conditional statement executed by the concrete execution, the symbolic execution generates symbolic constraints over the input variables. At the execution of an assertion, the symbolic execution engine invokes a theorem prover to check that the assertion is verified, according to the constraints encountered. If the assertion is not guaranteed, a violation of the behavior is reported. To evaluate their approach, the authors use 5 small and medium sized programs from SIR, including gzip, bc, hoc, space, and printtokens. The results show that they are able to detect buffer overflows but it needs optimization because of the huge overhead that the instrumentation add.

Mouelhi *et al.* [Mouelhi 2009] enhance tests oracles for access control logic, also called Policy Decision Point (PDP). This is done in 3 steps: select test cases that execute PDPs, map each of the test cases to specific PDPs and oracle enhancement. They add to the existing oracle checks that the access is granted or denied with respect to the rule and checks that the PDP is correctly called. To do so, they force the Policy Enforcement Point, *i.e.* the point where the policy decision is setting in the system functionality, to raise an exception when the access is denied and they compare the produced logs with expected log. To evaluate, they conduct case studies on three Java applications developed by students during group projects. For these three subjects, the number of classes ranges form 62 to 122, the number of methods ranges from 335 to 797, and the number of lines of code ranges from 3204 to 10703. The experimental results show that compared to manual testing, automated oracle generation saves a lot of time (from 32 hours to 5 minutes).

Daniel *et al.* [Daniel 2009b] devise *ReAssert* to automatically repair test cases, *i.e.* to modify test cases that fail due to a change. *ReAssert* follows five steps: record the values of failing assertions, re-executes the test and catch the failure exception, *i.e.* the exception thrown by the failing assertion. From the exception, it extracts the stack trace to find the code to repair. Then, it selects the repair strategy depending on the structure of the code and on the recorded value. Finally, ReAssert re-compiles the code changes and repeats all steps until no more assertions fail. The tool was evaluated on six real and well known open source Java projects, namely *PMD*, *JFreeChart*, *Lucene*, *Checkstyle*, *JDepend* and *XStream*. The authors created a collection of manually written and generated tests methods by targeting previous versions of these programs. *ReAssert* was able to produce fixes from 25% to 100% of failing tests for all study subjects. An usability study was also carried out with two teams of 18 researchers working on three research prototypes. The participants were asked to accomplish a number of tasks to write failing tests for new requirements and code changes and were also asked to manually fix the failures. *ReAssert* could repair 98% of failures created by the participants' code changes. In 90 % of cases the repairs suggested by the tool matched the patches created by the participants. The authors explain that the success rate of the tool depends more on the structure of the code of the test than the test failure itself.

2.5.3 Purification

Xuan *et al.* [Xuan 2016a] propose a technique to split existing tests into smaller parts in order to "purify" test methods. Here, purification can be seen as a form of test refactoring. A pure test executes one, and only one, branch of an if/then/else statement. On the contrary, an impure test executes both branches *then* and *else* of the same if/then/else statement in code. The authors evaluate their technique on 5 widely used open-source projects from code organizations such as Apache. The experimental results show that the technique increases the purity of test cases by up to 66% for if statements and 11% for try statement. In addition, the result also shows that the technique improves the effectiveness of program repair of Nopol [Xuan 2017].

Xuan *et al.* [Xuan 2014] aim at improving the fault localization capabilities by *purifying* test cases. By purifying, they mean to modify existing failing test methods into single assertion test cases and remove all statements that are not related to the assertion. They evaluated the test purification on 6 open-source java project, over 1800 bugs generated by typical mutation tool PIT and compare their results with 6 mature fault localization techniques. They show that they improve the fault localization effectiveness on 18 to 43% of all the faults, as measured per improved wasted effort.

2.5.4 Summary

Main achievements: What is remarkable in AMP_{mod} is the diversity of engineering goals considered. Input space exploration provides better state coverage [Dallmeier 2010] and reliability assessment [Hamlet 1993], oracle improvement allows to increase the efficiency and effectiveness of tests [Xie 2006, Carzaniga 2014, Joshi 2007, Mouelhi 2009, Daniel 2009b], test purification of test cases facilitate program repair [Xuan 2016a] and fault localization [Xuan 2014].

Main challenges: Although impressive results have been obtained, no experiments have been carried out to study the acceptability and maintainability of amplified tests [Dallmeier 2010, Xie 2006, Hamlet 1993, Carzaniga 2014, Joshi 2007, Mouelhi 2009, Daniel 2009b, Xuan 2016a, Xuan 2014]. In this context, acceptability means that human developers are ready to commit the amplified tests to the version control system (*e.g.* in the Git repository). The maintainability challenge is whether the machine-generated tests can be later understood and modified by developers.

2.6 Analysis

2.6.1 Aggregated View

Table 2.1 shows all the articles considered in this snowballing survey per the inclusion criteria. The first column of the table shows the citation information, as given in the "References" section. The second column shows the term that the authors use to designate the form of amplification that they investigate. Columns 3 to 18 are divided in three groups. The first group corresponds to the section in which the paper has been included. The second group corresponds to the different identified engineering goals . The third group captures the different techniques used for amplification in each work. The final columns in the table contain the target programming language, the year and venue in which the paper has been published, the last name of the first author and the iteration of the snowballing process in which the paper was included in the study.

Each row in the table corresponds to a specific contribution. The rows are sorted first by the section in which the papers are included in the study, then by year and then by the last name of the first author. In total, the table contains 49 rows.

One can see that "augmentation" (15 contributions), "generation" (9 contributions) and "amplification" (7 contributions) are the terms that appear most frequently to describe the approaches reported here. Other similar terms such as "enrichment", "adaptation" and "regeneration" are used less frequently. Most proposals (19 contributions) focus on adding new test cases to the existing test suite. Test amplification in the context of a change or the modification of existing test cases have received comparable attention (16 and 14

Table 2.1: List of papers included in this snowballing survey.

Reference	Term used	Add new tests	With respect to change/diff	Runtime modification	Modifies existing tests	Improve coverage	Reproduce crashes	Detect new faults	Localize faults	Improve repair	Improve observability	Test code analysis	Application code analysis	Execution modification	Concolic execution	Symbolic execution	Search based heuristics	Target language	Venue	Publication year	Last name of first author	Iteration
[Harder 2003]	augmentation	•						•				•						C	ICSE	2003	Harder	2
[Baudry 2002]	optimization	•				•						•					•	.NET	ASE	2002	Baudry	2
[Baudry 2005b]	optimization	•				•						•					•	Eiffel, C#	STVR	2005	Baudry	3
[Baudry 2005a]	optimization	•				•						•					•	C#	IEEE Software	2005	Baudry	3
[Pacheco 2005b]	generation	•						•				•						Java	ECOOP	2005	Pacheco	1
[Baudry 2006]	optimization	•							•			•						Java	ICSE	2006	Baudry	4
[Tillmann 2006]	generation	•			•	•						•				•		Spec#	IEEE Software	2006	Tillmann	5
[Marri 2010]	generalization	•			•	•						•				•		C#	FASE	2011	Thummalapenta	5
[Fraser 2011c]	generation	•						•				•	•					Java	ISSTA	2011	Fraser	6
[Röβler 2012]	generation	•							•			•						Java	ISSTA	2012	Ropler	5
[Yoo 2012]	regeneration	•				•						•					•	Java	STVR	2012	Yoo	4
[Pezze 2013]	generation	•						•				•						Java	ICST	2013	Pezze	5
[Yu 2013]	augmentation	•							•			•						Java	IST	2013	Yu	5
[Bloem 2014]	augmentation	•				•						•				•		C	QSIC	2014	Bloem	3
[Milani Fard 2014]	generation	•						•				•						JavaScript	ASE	2014	Fard	3
[Xuan 2015]	mutation	•					•					•						Java	ESEC/FSE	2015	Xuan	3
[Rojas 2016]	generation	•				•						•					•	Java	STVR	2016	Rojas	5
[Yoshida 2016]	augmentation	•			•	•						•				•		C, C++	ISSTA	2016	Yoshida	3
[Patrick 2017]	generation	•						•									•	C	IST	2017	Patrick	4
[Apiwattanapong 2006]	augmentation		•			•					•		•			•		Java	TAIC PART	2006	Apiwattanapong	3
[Santelices 2008]	augmentation		•			•					•		•			•		Java	ASE	2008	Santelices	3
[Daniel 2009b]	repairing refactoring		•		•						•	•						Java	ASE	2009	Daniel	4
[Xu 2009]	augmentation		•			•									•			Java	APSEC	2009	Xu	3
[Qi 2010]			•			•							•			•		C	ASE	2010	Qi	4
[Xu 2010a]	augmentation		•			•							•				•	Java	GECCO	2010	Xu	3
[Xu 2010b]	augmentation		•			•							•		•		•	C	FSE	2010	Xu	2
[Santelices 2011]	augmentation		•			•					•		•					Java	ICST	2011	Santelices	3
[Xu 2011]	augmentation		•			•							•		•		•	C	ISSRE	2011	Xu	3
[Mirzaaghaei 2012]	repairing adaptation		•		•	•				•		•	•					Java	ICST	2012	Mirzaaghaei	3
[Mirzaaghaei 2014]	repairing adaptation		•		•	•				•		•	•					Java	SVTR	2014	Mirzaaghaei	3
[Böhme 2013]			•			•		•					•			•	•	C	ESEC/FSE	2013	Böhme	3
[Marinescu 2013]			•			•							•			•	•	C	ESEC/FSE	2013	Marinescu	5
[Wang 2014]	augmentation		•			•					•		•			•		Java	CSTVA	2014	Wang	3
[Xu 2015]	augmentation		•			•							•		•		•	C	STVR	2015	Xu	3
[Palikareva 2016]			•			•					•		•			•		C	ICSE	2016	Palikareva	4
[Zhang 2012]	amplification			•				•					•	•				Java	ICSE	2012	Zhang	S
[Zhang 2014]	amplification			•				•					•	•				Java	TOSEM	2014	Zhang	
[Leung 2012]	amplification			•				•					•					CUDA	PLDI	2012	Leung	S
[Cornu 2015]	amplification			•				•		•			•	•				Java	IST	2015	Cornu	1
[Fang 2015]	amplification			•				•					•					Java	ECOOP	2015	Fang	1
[Zhang 2016]	augmentation			•		•		•			•		•			•		Java	FSE	2016	Zhang	3
[Hamlet 1993]	amplification				•						•								ISSTA	1993	Hamlet	S
[Xie 2006]	augmentation				•							•		•				Java	ECOOP	2006	Xie	5
[Joshi 2007]	amplification				•						•				•			C	ESEC/FSE	2007	Joshi	S
[Mouelhi 2009]					•						•	•						Java	ICST	2009	Mouelhi	4
[Dallmeier 2010]	enrichment				•						•	•						Java	ISSTA	2010	Dallmeier	4
[Carzaniga 2014]	cross-checking				•						•			•				Java	ICSE	2014	Carzaniga	6
[Xuan 2014]	purification				•				•			•						Java	FSE	2014	Xuan	5
[Xuan 2016a]	purification refactoring				•					•		•						Java	IST	2016	Xuan	1

contributions respectively). Some techniques that modify existing test cases also target the addition of new test cases (3 contributions) and amplify the test suite with respect to a change (3 contributions). Amplification by runtime modification is the least explored area.

Most works aim at improving the code coverage of the test suite (25 contributions). After that, the main goals are the detection of new faults and the improvement of observability (13 and 12 contributions respectively). Fault localization, repair improvement and crash reproduction receive less attention (4, 4 and 1 contributions respectively).

47 papers included in the table have been published between 2003 and 2017. One paper was published back in 1993. Between years 2009 and 2016 the number of papers has been stable (mostly four or five per year). In 2014 two extensions to previous works have been published in addition to five original works, making it the year with most publications on the subject.

Figure 2.1 visualizes the snowballing process. Every node of the graph corresponds to a reviewed paper. Seed papers are represented as filled rectangles to distinguish them from the rest. All nodes incorporated in the same iteration are clustered together. The edges shown in the graph correspond to the references followed to include the paper. Backward references are marked in green and labelled “B”. For these edges, the origin node cites the target node. Forward references are marked in blue and labelled “F”. For these edges, the origin node is cited by the target node.

2.6.2 Technical Aspects

Most works include some form of test or application code analysis (26 and 21 contributions respectively). Notably, the majority of works that add new test methods also include a test code analysis phase. All papers that amplify the test suite with respect to a change also include an application analysis stage. Search-based heuristics and symbolic execution are used to a large extent (12 contributions each), while concolic execution and execution modification are the least used techniques (5 contributions each).

Java programs are the most targeted systems (30 contributions), followed by C programs (12 contributions). JavaScript applications have received very little attention in the area (only one row).

2.6.3 Tools for Test Amplification

Most test case amplification papers discussed in this paper are experimental in nature, and are based on a prototype tool. For the field to mature, it is good if researchers can reproduce past results, and compare their new techniques against existing ones. To this extent, it feels that open-science in the form of publicly-available and usable research prototypes is of utmost importance.

Figure 2.1: Visualization of the snowballing process. Each node corresponds to a paper included in the study. Seed papers are differentiated form the rest. Papers added in the same iteration are clustered together. F blue edges represent forward references. B represent backward references.

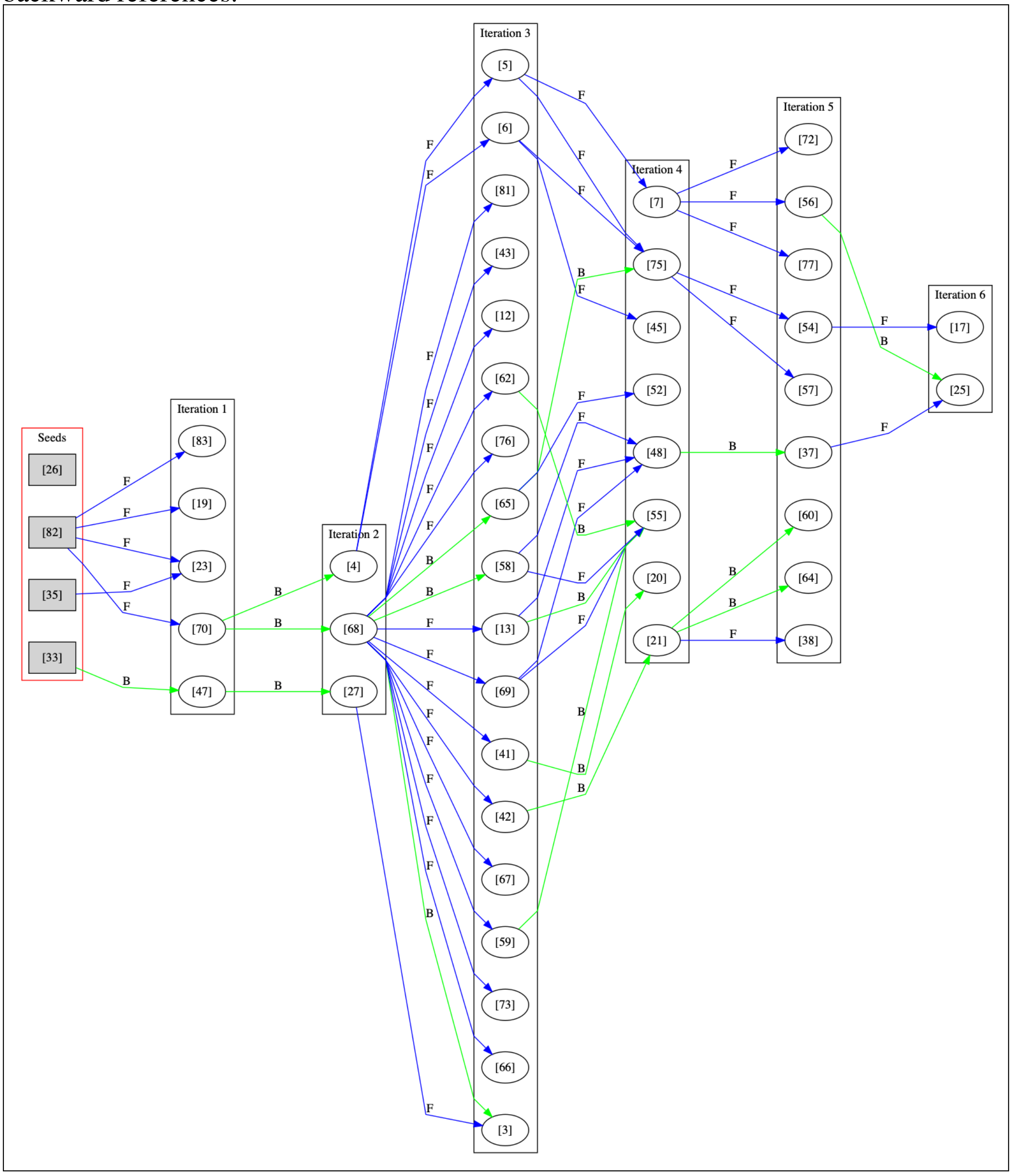


Table 2.2: List of surveyed papers in which a URL related to a tool has been found.

Reference	URL	Observations
[SIR]	http://sir.unl.edu	This is a software repository. It is not a tool for amplification but it is a resource that could be used for amplification.
[Baudry 2006]	http://www.irisa.fr/triskell/results/Diagnosis/index.htm	The URL points only to results.
[Böhme 2014]	http://www.comp.nus.edu.sg/~release/corebench/	The website also contains empirical results.
[Carzaniga 2014]	http://www.inf.usi.ch/phd/goffi/crosscheckingoracles/	
[Dallmeier 2010]	https://www.st.cs.uni-saarland.de/models/tautoko/index.html	
[Daniel 2009b]	http://mir.cs.illinois.edu/reassert/	
[Fang 2015]	https://bitbucket.org/fanglu/perfblower-public	There is no explicit url in the paper but a sentence saying that the tool is available in Bitbucket. With this information it was easy to find the URL.
[Fraser 2011b]	http://www.evosuite.org/	Additional materials included.
[Marri 2010]	https://sites.google.com/site/asergrp/projects/putstudy	The website also contains empirical results.
[Milani Fard 2014]	https://github.com/saltlab/Testilizer	
[Pacheco 2005b]	http://groups.csail.mit.edu/pag/eclat/	The website provides basic usage example.
[Palikareva 2016]	https://srg.doc.ic.ac.uk/projects/shadow/	The website also contains empirical results.
[Pezze 2013]	http://puremvc.org/	The paper has been turned into a company. The provided url is the url of this company.
[Röβler 2012]	https://www.st.cs.uni-saarland.de/bugex/	The url lives, but there is no way to download and try the tools.
[Xuan 2016a]	https://github.com/Spirals-Team/banana-refactoring	
[Xuan 2017]	https://github.com/SpoonLabs/nopol	Still active.
[Zhang 2016]	https://github.com/sei-pku/Ison	

With this in mind, the mentioned tools have been surveyed, if any. The protocol was as follows. First, looks for a URL in the paper, pointing to a web page containing the code of the tool or experimental data. For each URL, opens it in a browser between March 1st and March 31st 2018, to check that the page still exists and indeed contains experimental material.

Table 2.2 contains all valid URLs found. Overall, 17 valid open-science URLs have been identified. It may be considered as a low ratio, and thus calls for more open-science and reproducible research in the field of test amplification.

2.7 Conclusion

Test suites are now a mandatory component of serious software development. Developers spend time and effort developing these test suites, incorporating precious and unique knowledge inside it. These test suites are available to innovate new approaches and in the last decades, the research has been prolific.

In this state of the art, I showed that researchers have been inspired by these available test suites These research works achieve different goals such as improving the coverage of the test suite, faults detection, purification, etc. In one hand, I also showed that test amplification takes different form such as the generation of new test methods as variant of an existing test method or even the modification of the test execution. In the other hand, there are still challenges to be tackled as highlighted by this survey. For example, test amplification according to multiples changes or for several works. Also, I observed that among all of the reviewed works, only 17 out of 49 provides a link to tool or/and experimental data. This is a big issue for reproducibility and prevent the research community to build upon existing works.

Most important gaps are no experimentation has been conducted while including real developers in the loop and the lack of evidence that results can be generalized. Test amplification seems never been used to enhance regression testing capability of the test suite. In my opinion, test amplification is a good candidate to improve the ability of the test suite to detect regressions, or more generally detect a behavioral difference between two versions of the same program.

In this thesis, I aim at addressing these problems by providing a test suite amplification tool. The goal of this tool is to provide test methods that detect a behavioral change, introduced by a developer for example. In addition to this, the outcome of this tool must be generalizable to any software. This is why I conduct two kinds of experimentation: executing the tool on a large benchmark of real programs from open-source community. And obtain the assessment of real developers on the quality of this tool's output. This tool, called DSpot, is presented in the next chapter. DSpot has the ambition to be strong enough to be applied on an arbitrary program, to bring together approaches that have never be gathered before. All the data produced for empirical evaluation are also open-source in order to promote open-science and let the research community relies on open-data to pursue the effort in this emerging field that is test amplification.

CHAPTER 3

DSpot: A Test Amplification Technique

Contents

As mentioned in the introduction of the previous chapter (see Chapter 2), developers are mandated to build strong test suites in parallel of their application. This reinforces the confidence that they have in the correctness of their application.

However, test suites are not unfortunately the first objective and thus pass to the background for developers. Their development is focused on regular cases on the way that the program should behave. In addition to this, developers may cut corners because of lack of time, expertise or discipline.

In the literature, one can find a lot of work trying to solve this issue, such as test suite generation or test suite evolution [McMinn 2004, Edvardsson 2002]. Test amplification [Yoo 2012, Danglot 2019b] is one of them. However, the surveyed works present problems (see subsection 2.2.5, subsection 2.3.4, subsection 2.4.2, subsection 2.5.4), in particular generalization of results, test amplification according to multiple changes, scaling-up and assessment of the result by external real developers.

In this chapter, I expose the major output of this thesis: DSpot. DSpot is a test amplification tool that has the ambition to improve the test suite of real projects. DSpot achieves this by providing a set of automated procedures done in three majors steps:

1. it modifies the test inputs in order to trigger new behaviors.
2. it generates assertions to verify the new behavior of the program.
3. it selects amplified test methods according to a specific test-criterion such as branch coverage.

DSpot's output is a set of amplified test methods that improve the original test suite according to the specified test-criterion. The goal is to suggest these amplified test methods to the developers as possible improvements of the test suite. For example, I could use the pull-request mechanisms to seamlessly suggest these improvements to developers, as part of their regular development activities.

In this chapter, I first define key concepts in Section 3.1. Then, I expose an overview of DSpot with its principle, input& output, and its workflow in Section 3.2. Followed by the explanation of DSpot's algorithm in Section 3.3. Then, I detail the implementation and the ecosystem of DSpot in Section 3.4 Eventually, I conclude this chapter in Section 3.5.

3.1 Definitions

I first define the core terminology of DSpot in the context of object-oriented Java programs.

Test suite is a set of test classes.

Test class is a class that contains test methods. A test class is neither deployed nor executed in production.

Test method or **test case** is a method that sets up the system under test into a specific state and checks that the actual state at the end of the method execution is the expected state.

Unit test is a test method that specifies a targeted behavior of a program. Unit tests are usually independent from each other and execute a small portion of the code, *i.e.* a single unit or a single component of the whole system.

System test or **Integration test** is a test method that specifies a large and complex behavior of a program. System tests are usually large and use a lot of different components of the program.

Test-criterion is a measure of the quality of the test suite according to an engineering goal. For instance, one can measure the execution speed of its test suite, and consider that the faster it is executed the better it is. The most popular is probably the execution coverage, which can be measured at different level: branches, statements, instructions. It measures the proportion of the program that the test suite executes. The larger is this proportion, the better is considered the test suite since it is likely to verify more behavior.

Test inputs are the first key component of test methods. The input setup part is responsible for driving the program into a specific state. For instance, one creates objects and invokes methods on them to produce a specific state.

Assertions are the second key component of test methods. The assertion part is responsible for assessing that the actual behavior of the program corresponds to the expected behavior, the latter being called the oracle. To do so, the assertion uses the state of the program, *i.e.* all the observable values of the program, and compare it to expected values, usually hard-coded by developers. If the actual observed values of the program state and the oracle are different (or if an exception is thrown), the test fails and the program is considered as incorrect.

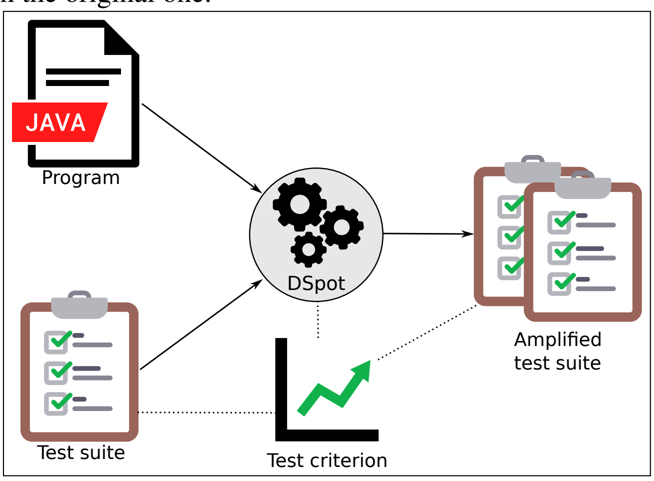
Amplified test suite is an existing test suite to which amplified test methods has been added.

Amplified test method is a test method that has been amplified, *i.e.* it has been obtained using an test amplification process and an existing test method.

3.2 Overview

3.2.1 Principle

Figure 3.1: DSpot's principle: DSpot takes as input a program, an existing test suite, and a test-criterion. DSpot outputs a set of amplified test methods. When added to the existing test suite, these amplified test methods increase the test-criterion, *i.e.* the amplified test suite is better than the original one.



DSpot is a test amplification tool. Its goal is to improve an existing test suite according to a specific test-criterion. DSpot takes as input the program, an existing test suite, and a test-criterion. The output of DSpot is a set of amplified test methods that are variants of existing test methods. When added to the existing test suite, it create an amplified test suite. This amplified test suite is better than the original test suite according to the test-criterion used during the amplification. For instance, one amplifies its test suite using branch coverage as test-criterion. This amplified test suite will execute more branches than the exiting test suite, *i.e.* the one without amplified test methods. In DSpot there are for now 3 test-criterion available: 1) keeping amplified test methods that increase the mutation score; 2) keeping amplified test methods that increase the instruction coverage; 3) keeping amplified test methods that detect the behavioral difference between two versions of the same program.

Figure 3.1 shows graphically the principle of DSpot.

3.2.2 Input & Output

DSpot's inputs are a program, a set of existing test methods and a test-criterion. The program is used as ground truth: in DSpot we consider the program used during the amplification correct. The existing test methods are used as a seed for the amplification. DSpot applies transformation individually to these test methods in order to improve the overall quality of the test suite with respect to the specified test-criterion.

DSpot produces variants of the test methods provided as input. These variants are called amplified test methods, since there are test methods that has been obtained using an amplification process. These amplified test methods are meant to be added to the test suite. By adding amplified test methods to the existing test suite, it creates an amplified test suite that improves the overall test suite quality. By construction, the amplified test suite is better than the original one with respect to the specified criterion.

An amplified test method's integration can be done in two way: 1) the developer integrates as it is the amplified test method into the test suite; 2) the developer integrate only the changes between the original test method and the amplified test method. This enrich directly an existing test method.

Listing 3.1 shows an example of changes' set obtained using DSpot.

By construction, all DSpot's amplification can be represented as a diff on an existing test method since amplified test methods are variants of existing ones.

3.2.3 Workflow

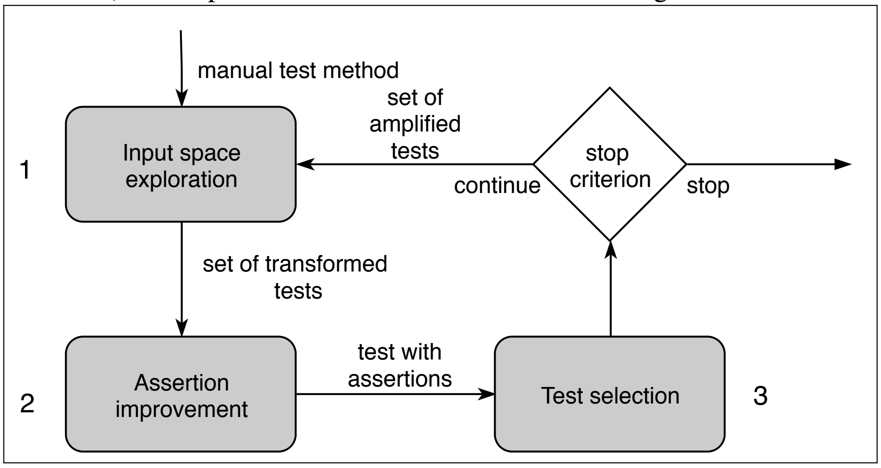
The main workflow of DSpot is composed of 3 main phases: 1) the modification of test code's inputs inspired by Tonella's technique [Tonella 2004], called "input space explo-

Listing 3.1: Example of what DSpot produces: a diff to improve an existing test case.

```
@@ -144,7 +144,8 @@ public void testEmptyList() throws Exception
     ArrayList<Foo> foos = new ArrayList<Foo>();

     ByteArrayOutputStream out = new ByteArrayOutputStream();
-    writeListTo(out, foos, SerializableObjects.foo.cachedSchema());
+    final int bytesWritten =
+      writeListTo(out, foos, SerializableObjects.foo.cachedSchema()
+    );
+    assertEquals(0, bytesWritten);
     byte[] data = out.toByteArray();
```

Figure 3.2: DSpot's workflow in three main steps: 1) the modification of test code's inputs, called "input space exploration"; 2) the addition of new assertions called "assertion improvement"; 3) the amplified test methods selection according to a test-criterion.



ration"; this phase consists in modifying test values (*e.g.* literals), objects and methods calls, the underlying details will be explained in subsection 3.3.1; 2) the addition of new assertions per Xie's technique [Xie 2006], this phase is called "assertion improvement" The behavior of the system under test is considered as the oracle of the assertion, see subsection 3.3.2. In DSpot, the combination of both techniques, *i.e.* the combination of input space exploration and assertion improvement is called "test amplification"; 3) the amplified test methods selection according to a given test-criterion, *e.g.* branch coverage. Eventually, DSpot either stops or continues to apply test amplification, according to a pre-defined stop-criterion. By doing this, DSpot stacks the transformation of test methods. In other words, DSpot amplifies already amplified test methods, which is possible because DSpot's output are real test methods.

In DSpot, the used stop-criterion is a number of iteration. However, one can imagine others kinds of stop-criterion such as a time budget, a test-criterion goal(*e.g.* reach 50% of mutation score) or a finite number of amplified test methods.

Listing 3.2: An example of an object-oriented test case (inspired from Apache Commons Collections)

```
1  testIterationOrder() {
2   // contract: the iteration order is the same as the insertion
       order
3
4   TreeList tl=new TreeList();
5   tl.add(1);
6   tl.add(2);
7
8   ListIterator it = tl.listIterator();
9
10  // assertions
11  assertEquals(1, it.next().intValue());
12  assertEquals(2, it.next().intValue());
13 }
```

3.2.4 Test method example

DSpot amplifies Java program's test methods, which are typically composed of two parts: test inputs and assertions, see Section 3.1.

Listing 3.2 illustrates an archetypal example of such a test case: first, from line 4 to line 6, the test input is created through a sequence of object creations and method calls; then, at line 8, the tested behavior is actually triggered; the last part of the test case at 11 and 12, the assertion part, specifies and checks the conformance of the observed behavior with the expected one. Note that this notion of call sequence and complex objects is different from test inputs consisting only of primitive values.

3.2.4.1 Best target test

By the algorithm's nature, unit tests (vs integration test) are the best target for DSpot. The reasons are behind the very nature of unit tests: First, they have a small scope, which allow DSpot to intensify its search while an integration test, that contains a lot of code, would make DSpot explore the neighborhood in different ways. Second, that is a consequence of the first, the unit tests are fast to be executed against integration test. Since DSpot needs to execute multiple times the tests under amplification, it means that DSpot would be executed faster when it amplifies unit tests than when it amplified integration tests.

Table 3.1: Literal test transformations in DSpot

Types	Operators
Number	add 1 to an integer
	minus 1 to an integer
	replace an integer by zero
	replace an integer by the maximum value (Integer.MAX_VALUE in Java)
	replace an integer by the minimum value (`Integer.MIN_VALUE` in Java).
Boolean	negate the value.
String	replace a string with another existing string.
	replace a string with white space, or a system path separator, or a system file separator.
	add 1 random character to the string.
	remove 1 random character from the string.
	replace 1 random character in the string by another random character.
	replace the string with a random string of the same size.
	replace the string with the `null` value.

3.3 Algorithm

3.3.1 Input Space Exploration Algorithm

DSpot aims at exploring the input space so as to set the program in new, never explored states. To do so, DSpot applies code transformations to the original manually-written test methods. ***I-Amplification*** for Input Amplification, is the process of automatically creating new test input points from existing test input points. DSpot uses three kinds of *I-Amplification*:

1) Amplification of literals: the new input point is obtained by changing a literal used in the test (numeric, boolean, string). These transformations are summarized in subsection 3.3.1.

2) Amplification of method calls: DSpot manipulates method calls as follows: DSpot duplicates an existing method call; removes a method call; or adds a new invocation to an accessible method with an existing variable as target.

3) Test objects: if a new object is needed as a parameter while amplifying method calls, DSpot creates an object of the correct type. In the same way, when a new method call needs primitive value parameters, DSpot generates a random value.

For example, if an *I-Amplification* is applied on the example presented in Listing 3.2, it may generate a new method call on *tl*. In Listing 3.3, the added method call is “removeAll”. During this process, DSpot removes existing assertions since they might fail because it changes the state of the program.

Listing 3.3: An example of an *I-Amplification*: the amplification added a method call to *removeAll()* on *tl*.

```
testIterationOrder() {
 TreeList tl=new TreeList();
 tl.add(1);
 tl.add(2);
 tl.removeAll(); // method call added

 // removed assertions
}
```

At each iteration, DSpot applies all kinds of *I-Amplification*, resulting in a set of input-amplified test methods. From one iteration to another, DSpot reuses the previously amplified tests, and further applies *I-Amplification*. By doing this, DSpot explore more the input space. The more iteration DSpot does, the more it explores, the more it takes time to complete.

3.3.2 Assertion Improvement Algorithm

A-Amplification: for Assertion Amplification, is the process of automatically creating new assertions. In DSpot, assertions are added on objects from the original test case, as follows: 1) it instruments the test methods to collect the state of a program after execution (but before the assertions), *i.e.* it creates observation points. The state is defined by all values returned by getter methods. 2) it runs the instrumented test to collect the values. This execution result in a map per test method, that gives the values from all getters. 3) it generates new assertions in place of the observation points, using the collected values as oracle. In addition, when a new test input sets the program in a state that throws an exception, DSpot produces a test asserting that the program throws a specific exception.

Listing 3.4: In *A-Amplification*, the second step is to instrument and run the test to collect runtime values.

```
testIterationOrder() {
 TreeList tl=new TreeList();
 tl.add(1);
 tl.add(2);aampl
 tl.removeAll();

 Observations.observe(tl.size());  // logging current behavior
 Observations.observe(tl.isEmpty());
}
```

For example, let consider *A-Amplification* on the test method of the example above. First, in Listing 3.4 DSpot instruments the test method to collect values, by adding method calls to the objects involved in the test case. Second, the test with the added observation points is executed, and subsequently, DSpot generates new assertions based on the collected values. In Listing 3.5, DSpot has generated two new assertions.

Listing 3.5: In *A-Amplification*, the last step is to generate the assertions based on the collected values.

```
testIterationOrder() {
 TreeList tl=new TreeList();
 tl.add(1);
 tl.add(2);
 tl.removeAll();

 // generated assertions
 assertEquals(0, tl.size()); // generated assertions
 assertTrue(tl.isEmpty()); // generated assertions
}
```

3.3.3 Pseudo-algorithm

Algorithm 1 shows the main loop of DSpot. DSpot takes as input a program P, its test suite TS and a test-criterion TC. DSpot also uses an integer n that defines the number of iterations and a set of input-amplifiers amp. DSpot produces an amplified test suite ATS, *i.e.* a better version of the input test suite TS according to the specified test criterion TC. First, DSpot initializes an empty set of amplified test methods ATS that will be outputted (Line 1). For each test case t in the test suite TS (Line 2), DSpot first tries to add assertions without generating any new test input (Line 3), method $generateAssertions\,(t)$ is explained in subsection 3.3.2. It adds to ATS the tests that improve the test-criterion(Line 4).

Note that adding missing assertions is the elementary way to improve existing tests. Consequently, in DSpot there are two modes, depending on the configuration:

1) DSpot executes only assertion amplification, if $n = 0$ or $amp = \emptyset$:

2) DSpot executes both input space exploration and assertion amplification, if $n > 0$ and $amp \neq \emptyset$

In the former mode, there is no exploration of the input space, resulting in a quick execution but less potential to improve the test-criterion. In the latter mode, the exploration, depending on n, takes times but have more potential to improve the test-criterion.

Algorithm 1 Main amplification loop of DSpot.

Require: Program P
Require: Test suite TS
Require: Test criterion TC
Require: Input-amplifiers $amps$ to generate new test data input
Require: n number of iterations of DSpot's main loop
Ensure: An amplified test suite ATS

```
1:  ATS ← ∅
2:  for t in TS do
3:      U ← generateAssertions(t)
4:      ATS ← {x ∈ U | x improves TC}
5:      TMP ← ATS
6:      for i = 0 to n do
7:          V ← []
8:          for amp in amps do
9:              V ← V ∪ amp.apply(TMP)
10:         end for
11:         V ← generateAssertions(V)
12:         ATS ← ATS ∪ {x ∈ V | x improves TC}
13:         TMP ← V
14:     end for
15: end forreturn ATS
```

DSpot initializes a temporary list of tests TMP with elements from ATS, if any (Line 5). Then it applies n times the following steps (Line 6): 1) it applies each amplifier amp on each tests of TMP to build V (Line 8-9 see subsection 3.3.1 *i.e.* *I-Amplification*); 2) it generates assertions on generated tests in V (Line 11 see subsection 3.3.2, *i.e.* *A-Amplification*); 3) it keeps the tests that improve the test-criterion(Line 12). 4) it assigns V to TMP for the next iteration. This is done because even if some amplified test methods in V have not been selected, it can contain amplified test methods that will eventually be better in subsequent iterations.

3.3.4 Flaky tests elimination

The input space exploration (see subsection 3.3.1) may produce test inputs that results in non-deterministic executions. This means that, between two independent executions, the state of the program is not the same. Since DSpot generates assertions where the expected value is a hard coded value from a specific run (see subsection 3.3.2), the generated test case may become flaky: it passes or fails depending on the execution and whether the expected value is obtained or not.

To avoid such flaky tests, DSpot run f times each new test case resulting from amplification (f = 3 in the default configuration). If a test fails at least once, DSpot throws it away.

This procedure does not guarantee the absence of flakiness. However, it gives incremental confidence: if the user wants more confidence, she can tell DSpot to run the amplified tests more times.

3.4 Implementation

DSpot is implemented in Java. It consists of 19295+ logical lines of code (as measured by cloc). DSpot uses Spoon[Pawlak 2015] to analyze and transform the tests of the software application under amplification.

3.4.1 Ecosystem

For the sake of open-science, DSpot is made publicly available on GitHub[1]. This repository is animated by the community around DSpot. It uses a pull-request based development to promote open-source contributions.

Since DSpot has been developed with the ultimate goal to serve developers in their task of testing their programs, I participated to the development of a rich ecosystem.

First, DSpot-maven is a maven plugin that allows developers to execute DSpot on their maven project without downloading anything. This plugin allows also developers to configure DSpot inside their own pom with specific setup in order to automate the application of DSpot.

Second, STAMP's partners developed an Eclipse plugin and Jenkins plugin. The former allows developers to run DSpot inside Eclipse, with a friendly UI to configure it. The latter allows developers to run DSpot as a Jenkins jobs in order to integrate DSpot in their continuous integration service.

3.5 Conclusion

This chapter presented technical details about DSpot. DSpot is a test amplification tool that improves the test suite. DSpot is the implementation of the first algorithm that combines 2 approaches from the state of the art: the input modification by Tonella *et al.* [Tonella 2004] and the assertions generation by Xie[Xie 2006]. DSpot provides a new tool to improvement test methods automatically. It has the potential to assist developers in the extremely important task that is testing the application. DSpot's output is a set of amplified test methods that improve the original test suite according to the specified test-criterion. DSpot's algorithm is done in 3 main steps:

[1] https://github.com/STAMP-project/dspot

1. It transforms the input of tests using static analysis and code transformation to create a new state of the program.

2. it observes the new state of the program and generate assertions in order to obtain complete test methods.

3. it uses a test-criterion to keep only amplified test methods that improve the test suite with respect to this same criterion, *e.g.* branch coverage.

In the two following chapters, I evaluate the performance of DSpot to improve existing test suite in two scenarios: A first scenario where DSpot improves existing test suites of open-source projects from GitHub. DSpot's output is evaluated by external developers and the test-criterion to improve is the mutation score. A second scenario where DSpot is enhanced to be executed inside the continuous integration. The goal is to detect a behavioral changes introduced by commits done by developers on a version control platform such as GitHub.

Chapter 4

Test Amplification For Artificial Behavioral Changes Detection Improvement

Contents

This thesis aims at supporting developers in their developing tasks. One of them is to evolve the test suite, *i.e.* modify or add test methods, to strengthen their confidence in the program's correctness. In Chapter 3, I introduced DSpot which is a test suite amplification tool. In this chapter, I evaluate the effectiveness of DSpot to improve the quality of a test suite and the acceptability of the resulting amplified test methods.

This evaluation is based on the mutation score as test-criterion. I confronted DSpot's output to real projects from GitHub. To do so, I proposed to developers to integrate directly the amplified test methods into their test suite. Developers showed their interest in amplified test methods by permanently accepting some DSpot's amplified test methods into their test suite. I also performed an evaluation on 40 test classes from 10 projects from GitHub and showed that DSpot improves 26 of them.

To sum up, the contributions of this chapter are:

- the design and execution of an experiment to assess the relevance of DSpot, based on feedback from the developers of mature and active projects;
- the design and execution of a large scale quantitative study of the improvement of 40 real-world test classes taken from 10 mature open-source Java projects.
- fully open-science data: the experimental data are made publicly available for future research[1]

Note that this chapter has been published [Danglot 2019c] in the Springer journal *Empirical Software Engineering* and the remainder is as follows: Section 4.1 introduces mutation score as a test-criterion. Section 4.2 presents the experimental protocol of our study. Section 4.3 analyses our empirical results.. Section 4.4 discusses the threats to validity. and Section 4.5 concludes this chapter.

4.1 Mutation score as test-criterion

Mutation score measures the test suite's ability to detect artificial behavioral changes. Briefly, it is measured as follow:

1) it injects a fault, or an artificial behavioral change, in the source code, *e.g.* changes a $\geq$ into a $>$. This modified program is called "mutants". It generates different mutants with different artificial behavioral change;

2) it executes the test suite on the mutant;

3) it collects the result of the test suite execution. If at least on test method fails, it means that the test suite is able to detect the fault. It is said that the test suite kills the mutant; If no test methods failed, it means that the test suite is not able to detect the fault. It is said that the mutant remains alive.

4) to compute the mutation score, one must compute the percentage of mutants killed over the mutants generated. The more mutants the test suite kills, the better is considered the test suite.

Mutation score aims at emulating faults that a developer could integrate in his code. If the test suite has a high mutation score, the probability that it detects such fault increase.

DSpot uses Pitest [Coles 2016] [2] because:

1) it targets Java programs;

[1] https://github.com/STAMP-project/dspot-experiments/

[2] latest version released at the time of the experimentation: 1.2.0.https://github.com/hcoles/pitest/releases/tag/1.2.0

2) it is mature and well-regarded;

3) it has an active community.

The most important feature of Pitest is that if the application code remains unchanged, the generated mutants are always the same. This property is very interesting for test amplification. Since DSpot only modifies test code, this feature allows us to compare the mutation score of the original test method against the mutation score of the amplified version and even compare the absolute number of mutants killed by two test method variants. DSpot exploits this feature to use mutation score as a reliable test-criterion: since DSpot never modifies the application code, the set of mutants is the same between runs and thus allow DSpot to a concrete and stable baseline for the baseline. DSpot can compare mutants killed before and mutants killed after the amplification in order to select amplified test methods that kill mutants that were not killed by the original test suite.

By default, DSpot uses all the mutation operators available in Pitest: conditionals boundary mutator; increments mutator; invert negatives mutator; math mutator; negate conditionals mutator; return values mutator; void method calls mutator. For more information, see the dedicated section of Pitest's website: `http://pitest.org/quickstart/mutators/`.

In this experimentation, mutation score has been choose over coverage because mutation score is consider stronger than coverage. The purpose of test suites is to check the program's behavior. In one hand, coverage is only based on the execution of the program and do not require any oracles. Coverage does not measure the proportion of the behavior tested but only the proportion of code executed. In the other hand, mutation score requires oracles and thus to have a high mutation score, the test suite must contains oracles.

4.2 Experimental Protocol

Recalling that DSpot is a automatic test improvement process. Such processes have been evaluated with respect to evolutionary test inputs [Tonella 2004] and new assertions [Xie 2006]. However:

1) the two topics have never been studied in conjunction

2) they have never been studied on large modern Java programs

3) most importantly, the quality of improved tests has never been assessed by developers.

I set up a novel experimental protocol that addresses those three points. First, the experiment is based on DSpot, which combines test input exploration and assertion generation. Second, the experiment is made on 10 active GitHub projects. Third, I have proposed improved tests to developers under the form of pull-requests.

The evaluation aims at answering the following research questions:

4.2.1 Research Questions

RQ1: Are the improved test cases produced by DSpot relevant for developers? Are the developers ready to permanently accept the improved test cases into the test repository?
RQ2: To what extent are improved test methods considered as focused?
RQ3: To what extent do the improved test classes increase the mutation score of the original, manually-written, test classes?
RQ4: What is the relative contribution of *I-Amplification* and *A-Amplification* to the effectiveness of automatic test improvement?

4.2.2 Dataset

DSpot has been evaluated by amplifying test classes of large-scale, notable, open-source projects. The dataset includes projects that fulfil the following criteria:

1) the project must be written in Java;

2) the project must have a test suite based on JUnit;

3) the project must be compiled and tested with Maven;

4) the project must have an active community as defined by the presence of pull requests on GitHub, see subsection 4.3.1.

Table 4.1: Dataset of 10 active GitHub projects considered for experiments.

Project	Description	# LOC	# PR	Considered test classes
javapoet	Java source file generator	3150	93	TypeNameTest^h NameAllocatorTest^h FieldSpecTest^l ParameterSpecTest^l
mybatis-3	Object-relational mapping framework	20683	288	MetaClassTest^h ParameterExpressionTest^h WrongNamespacesTest^l WrongMapperTest^l
traccar	Server for GPS tracking devices	32648	373	GeolocationProviderTest^h MiscFormatterTest^h ObdDecoderTest^l At2000ProtocolDecoderTest^l
stream-lib	Library for summarizing data in streams	4767	21	TestLookup3Hash^h TestDoublyLinkedList^h TestICardinality^l TestMurmurHash^l
mustache.java	Web application templating system	3166	11	ArraysIndexesTest^h ClasspathResolverTest^h ConcurrencyTest^l AbstractClassTest^l
twilio-java	Library for communicating REST API	54423	87	RequestTest^h PrefixedCollapsibleMapTest^h AllTimeTest^l DailyTest^l
jsoup	HTML parser	10925	72	TokenQueueTest^h CharacterReaderTest^h AttributeTest^l AttributesTest^h
protostuff	Data serialization library	4700	35	TailDelimiterTest^h LinkBufferTest^h CodedDataInputTest^l CodedInputTest^h
logback	Logging framework	15490	104	FileNamePatternTest^h SyslogAppenderBaseTest^h FileAppenderResilience_AS_ROOT_Test^l Basic^l
retrofit	HTTP client for Android.	2743	249	RequestBuilderAndroidTest^h CallAdapterTest^h ExecutorCallAdapterFactoryTest^h CallTest^h

Those criteria have been implemented as a query on top of TravisTorrent [Beller 2017]. 10 projects has been selected from the result of the query which composed the dataset presented in Table 4.1. This table gives the project name, a short description, the number of pull-requests on GitHub (#PR), and the considered test classes. For instance, *javapoet* is a strongly-tested and active project, which implements a Java file generator, it has had 93 pull-requests in 2016.

4.2.3 Test Case Selection Process

For each project, 4 test classes have been select to be amplified. Those test classes are chosen as follows.

First, the test class must be a unit-test classes only, because DSpot focuses on unit test amplification. I use the following heuristic to discriminate unit test cases from others: test classes kept are test classes which executes less than an arbitrary threshold of S statements, *i.e.* if it covers a small portion of the code. In this experiment, $S = 1500$.

Among the unit-tests, 4 classes has been selected as follows. Since I want to analyze the performance of DSpot when it is provided with both good and bad tests, selected test classes has been split into two groups: one group with strong tests, one other group with low quality tests. mutation score has been used to distinguish between good and bad test classes. Accordingly, the selection process has five steps:

1) Compute the original mutation score of each class with Pitest (see Section 4.1;

2) Discard test classes that have 100% mutation score, because they can already be considered as perfect tests (this is the case for eleven classes, showing that the considered projects in the dataset are really well-tested projects);

3) Sort the classes by mutation score (see subsection 4.2.4), in ascending order;

4) Split the set of test classes into two groups: high mutation score ($> 50\%$) and low mutation score ($< 50\%$);

5) Randomly select 2 test classes in each group.

This selection results with 40 test classes: 24 in high mutation group score and 16 in low mutation score group. The imbalance is due to the fact that there are three projects really well tested for which there are none or a single test class with a low mutation score (projects protostuff, jsoup, retrofit). Consequently, those three projects are represented with 3 or 4 well-tested classes (and 1 or 0 poorly-tested class). In Table 4.1, the last column contains the name of the selected test classes. Each test class name is indexed by a "h" or a "l" which means respectively that the class have a high mutation score or a low mutation score.

4.2.4 Metrics

Number of Killed Mutants ($\#Killed.Mutants$): is the absolute number of mutants killed by a test class. It used to compare the fault detection power of an original test class and the one of its amplified version.

Mutation Score: is the percentage of killed mutants over the number of executed mutants. Mathematically, it is computed as follow:

$$\frac{\#Killed.Mutants}{\#Exec.Mutants}$$

.

Increase Killed: is the relative increase of the number of killed mutants by an original test class T and the number of killed mutants by its amplified version T_a. It is computed as follows:

$$\frac{\#Killed.Mutants_{T_a} - \#Killed.Mutants_T}{\#Killed.Mutants_T}$$

The goal of DSpot is to improve tests such that the number of killed mutants increases.

4.2.5 Methodology

This experimental protocol has been designed to study to what extent DSpot and its result are valuable for the developer.

- **RQ1** To answer to RQ1, pull-requests have been created on notable open-source projects. DSpot amplifies 19 test classes of selected projects and I propose amplified test methods to the main developers of each project under consideration in the form of pull requests (PR) on GitHub. A PR is composed of a title, a short text that describes the purpose of changes and a set of code change (aka a patch). The main developers review, discuss and decide to merge or not each pull request. I base the answer on the subjective and expert assessment from projects' developers. If a developer merges an improvement synthesized by DSpot, it validates the relevance of DSpot. The more developers accept and merge test improvements produced by DSpot into their test suite, the more the amplification is considered successful.

- **RQ2** To answer RQ2, the number of suggested improvements is computed, to verify that the developer is not overwhelmed with suggestions. The number of focused amplified test methods is computed following the technique described in subsubsection 4.3.1.2, for each project in the benchmark. I present and discuss the proportion of focused tests out of all proposed amplified tests.

- **RQ3** To answer RQ3, I see whether the value that is taken as proxy to the developer value – the mutation score– is appropriately improved. For 40 real-world classes, first Pitest (see Section 4.1) is ran the mutation testing tool on the test class. This gives the number of killed mutants for this original class. Then, the test class under consideration is amplified and the new number of killed mutants after amplification is computed. Finally, the result are compared and analyzed.

- **RQ4** To answer RQ4, the number of *A-Amplification* and *I-Amplification* amplifications are computed. The former means that the suggested improvement is very short hence easy to be accepted by the developer while the latter means that more time would be required to understand the improvement. First, I collect three series of metrics:

  1) I compute number of killed mutants for the original test class;

  2) I improve the test class under consideration using only *A-Amplification* and compute the new number of killed mutants after amplification;

  3) I improve the test class under consideration using *I-Amplification* as well as *A-Amplification* (the standard complete DSpot workflow) and compute the number of killed mutants after amplification.

  Then, I compare the increase of mutation score obtained by using *A-Amplification* only and *A-Amplification* + *I-Amplification*.[3]

Research questions 3 and 4 focus on the mutation score to assess the value of amplified test methods. This experimental design choice is guided by the approach to select "focused" test methods, which are likely to be selected by the developers (described in subsubsection 4.3.1.2). Recall that the number of killed mutants by the amplified test is the key focus indicator. Hence, the more DSpot is able to improve the mutation score, the more likely there are good candidates for the developers.

4.3 Experimental Results

4.3.1 Answer to RQ1

RQ1: Would developers be ready to permanently accept automatically improved test cases into the test repository?

4.3.1.1 Process

In this research question, the goal is to propose a new test to the lead developers of the open-source projects under consideration. The improved test is proposed through a "pull-request", which is a way to reach developers with patches on collaborative development platforms such as GitHub.

In practice, short pull requests (*i.e.* with small test modifications) with clear purpose, *i.e.* what for it has been opened, have much more chance of being reviewed, discussed and

[3]Note that the relative contribution of *I-Amplification* cannot be evaluated alone, because as soon as DSpot modifies the inputs in a test case, it is also necessary to change and improve the oracle (which is the role of *A-Amplification*).

eventually merged. So the goal is to provide improved tests which are easy to review. As shown in subsection 3.3.1, DSpot generates several amplified test cases, and all of them cannot be proposed to the developers. To select the new test case to be proposed as a pull request, I look for an amplified test that kills mutants located in the same method. From the developer's viewpoint, it means that the intention of the test is clear: it specifies the behavior provided by a given method or block.

4.3.1.2 Selection Of Amplified Method For Pull Requests

DSpot sometimes produces many tests, from one initial test. Due to limited time, the developer needs to focus on the most interesting ones. To select the test methods that are the most likely to be merged in the code base, the following heuristic is implemented: First, the amplified test methods are sorted according to the ratio of newly killed mutants and the total number of test modifications. Then, in case of equality, the methods are further sorted according to the maximum numbers of mutants killed in the same method.

The first criterion means that short modifications have more valuable than large ones. The second criterion means that the amplified test method is focused and tries to specify one specific method inside the code.

If an amplified test method is merged in the code base, the corresponding method is considered as specified. In that case, other amplified test methods that specify the same method are no longer taken into account.

Finally, in this ordered list, the developer is recommended the amplified tests that are focused, where focus is defined as where at least 50% of the newly killed mutants are located in a single method. The goal is to select amplified tests which intent can be easily grasped by the developer: the new test specifies the method.

For each selected method, I compute and minimize the diff between the original method and the amplified one and then the diff as a pull request is submitted. A second point in the preparation of the pull request relates to the length of the amplified test: once a test method has been selected as a candidate pull request, the diff is made as concise as possible for the review to be fast and easy.

4.3.1.3 Overview

In total, 19 pull requests has been created, as shown in subsubsection 4.3.1.3. In this table, the first column is the name of the project, the second is number of opened pull requests, *i.e.* the number of amplified test methods proposed to developers. The third column is the number of amplified test methods accepted by the developers and permanently integrated in their test suite. The fourth column is the number of amplified test methods rejected by the developers. The fifth column is the number of pull requests that are still being discussed,

Table 4.2: Overall result of the opened pull request built from result of DSpot.

Project	# opened	# merged	# closed	# under discussion
javapoet	4	4	0	0
mybatis-3	2	2	0	0
traccar	2	1	0	1
stream-lib	1	1	0	0
mustache	2	2	0	0
twilio	2	1	0	1
jsoup	2	1	1	0
prostostuff	2	2	0	0
logback	2	0	0	2
retrofit	0	0	0	0
total	19	14	1	4

i.e. nor merged nor closed. Note that these numbers might change over time if pull-requests are merged or closed.

Table 4.3: List of URLs to the pull-requests created in this experiment.

Project	Pull request URLs

javapoet	`https://github.com/square/javapoet/pull/669`
	`https://github.com/square/javapoet/pull/668`
	`https://github.com/square/javapoet/pull/667`
	`https://github.com/square/javapoet/pull/544`

mybatis-3	`https://github.com/mybatis/mybatis-3/pull/1331` `https://github.com/mybatis/mybatis-3/pull/912`
traccar	`https://github.com/traccar/traccar/pull/2897` `https://github.com/traccar/traccar/pull/4012`
stream-lib	`https://github.com/addthis/stream-lib/pull/128`
mustache	`https://github.com/spullara/mustache.java/pull/210` `https://github.com/spullara/mustache.java/pull/186`
twilio	`https://github.com/twilio/twilio-java/pull/437` `https://github.com/twilio/twilio-java/pull/334`
jsoup	`https://github.com/jhy/jsoup/pull/1110` `https://github.com/jhy/jsoup/pull/840`
protostuff	`https://github.com/protostuff/protostuff/pull/250` `https://github.com/protostuff/protostuff/pull/212`
logback	`https://github.com/qos-ch/logback/pull/424` `https://github.com/qos-ch/logback/pull/365`

Overall 14 over 19 have been merged. Only 1 has been rejected by developers. There are 4 under discussion. Table 4.3.1.3 contains the URLs of pull requests proposed in this experimentation.

In the following, one pull-request per project is analyzed.

4.3.1.4 javapoet

DSpot has been applied to amplify `TypeNameTest`. DSpot synthesizes a single assertion that kills 3 more mutants, all of them at line 197 of the equals method. A manual analysis reveals that this new assertion specifies a contract for the method `equals()` of objects of type `TypeName`: the method must return false when the input is null. This contract was not tested.

Consequently, I have proposed to the Javapoet developers one liner pull request [4] showed in Listing 4.1.

Listing 4.1: Test-improvement proposed to Javapoet developers.

```
@@ -178,5 +179,6 @@ private void assertEqualsHashCodeAndToString(TypeName a,
    TypeName b) {
  assertEquals(a.toString(), b.toString());
  assertThat(a.equals(b)).isTrue();
  assertThat(a.hashCode()).isEqualTo(b.hashCode());
+ assertFalse(a.equals(null));
```

The title of the pull resuest is: "*Improve test on TypeName*" with the following short text: "*Hello, I open this pull request to specify the line 197 in the equals() method of com.squareup.javapoet.TypeName. if (o == null) return false;*" This test improvement synthesized by DSpot has been merged by of the lead developer of javapoet one hour after its proposal.

4.3.1.5 mybatis-3

In project mybatis-3, DSpot has been applied to amplify a test for `MetaClass`. DSpot synthesizes a single assertion that kills 8 more mutants. All new mutants killed are located between lines 174 and 179, *i.e.* the `then` branch of an `if-statement` in method `buildProperty(String property, StringBuilder sb)` of `MetaClass`. This method builds a String that represents the property given as input. The `then` branch is responsible to build the String in case the `property` has a child, *e.g.* the input is "richType.richProperty". This behavior is not specified at all in the original test class.

I proposed to the developers the pull request, showed in Listing 4.2 entitled "*Improve test on MetaClass*" with the following short text: "*Hello, I open this pull request to specify the lines 174-179 in the buildProperty(String, StringBuilder) method of MetaClass.*", [5]

[4] https://github.com/square/javapoet/pull/544
[5] https://github.com/mybatis/mybatis-3/pull/912/files

Listing 4.2: Test-improvement proposed to MyBatis-3 developers.

```
@@ -65,6 +65,8 @@ public void shouldCheckGetterExistance() {
   assertTrue(meta.hasGetter("richType.richMap"));
   assertTrue(meta.hasGetter("richType.richList[0]"));

+  assertEquals(
+    richType.richProperty",
+    meta.findProperty("richType.richProperty", false)
+  );
```

The developer accepted the test improvement and merged the pull request the same day without a single objection.

4.3.1.6 traccar

DSpot has been applied to amplify `ObdDecoderTest`. It identifies a single assertion that kills 14 more mutants. All newly killed mutants are located between lines 60 to 80, *i.e.* in the method `decodesCodes()` of `ObdDecoder`, which is responsible to decode a `String`. In this case, the pull request consists of a new test method because the new assertions do not fit with the intent of existing tests. This new test method is proposed into `ObdDecoderTest`, which is the class under amplification. The PR was entitled "*Improve test cases on ObdDecoder*" with the following description: "*Hello, I open this pull request to specify the method decodeCodes of the ObdDecoder*". [6] The PR is shown in Listing 4.3.

Listing 4.3: Test-improvement proposed to traccar developers.

```
@@ -16,4 +16,10 @@ public void testDecode() {

   }

+   @Test
+   public void testDecodeCodes() throws Exception {
+     Assert.assertEquals("P0D14", ObdDecoder.decodeCodes("0D14").getValue());
+     Assert.assertEquals("dtcs", ObdDecoder.decodeCodes("0D14").getKey());
+   }
```

The developer of traccar thanked us for the proposed changes and merged it the same day.

4.3.1.7 stream-lib

DSpot has been applied to amplify `TestMurmurHash`. It identifies a new test input that kills 15 more mutants. All newly killed mutants are located in method `hash64()` of `MurmurHash` from lines 158 to 216. This method computes a hash for a given array of

[6] https://github.com/tananaev/traccar/pull/2897

byte. The PR, shown in Listing 4.4, was entitled "*Test: Specify hash64*" with the following description: "*The proposed change specifies what the good hash code must be. With the current test, any change in "hash" would still make the test pass, incl. the changes that would result in an inefficient hash.*". [7]

Listing 4.4: Test-improvement proposed to stream-lib developers.

```
@@ -44,7 +44,7 @@ public void testHash64ByteArrayOverload() {
   String input = "hashthis";
   byte[] inputBytes = input.getBytes();

- long hashOfString = MurmurHash.hash64(input);
+ long hashOfString = -8896273065425798843L;
   assertEquals("MurmurHash.hash64(byte[]) did not match MurmurHash.hash64(String)
        ",
   hashOfString, MurmurHash.hash64(inputBytes));
```

Two days later, one developer mentioned the fact that the test is verifying the overload of the method and is not specifying the method hash itself. He closed the PR because it was not relevant to put changes there. He suggested to open an new pull request with a new test method instead of changing the existing test method. I proposed, 6 days later, a second pull request entitled "*add test for hash() and hash64() against hard coded values*" with no description, since I estimated that the developer was aware of the test intention.[8]. This second pull request is shown in Listing 4.5.

The pull request has been merged by the same developer 20 days later.

4.3.1.8 mustache.java

DSpot has been applied to amplify `AbstractClassTest`. It identifies a try/catch/fail block that kills 2 more mutants. This is an interesting new case, compared to the ones previously discussed, because it is about the specification of exceptions, *i.e.* of behavior under erroneous inputs. All newly killed mutants are located in method `compile()` on line 194. The test specifies that if a variable is improperly closed, the program must throw a `MustacheException`. In the Mustache template language, an improperly closed variable occurs when an opening brace "{" does not have its matching closing brace such as in the input of the proposed changes. I propose the pull request, shown Listing 4.6, to the developers, entitled "*Add Test: improperly closed variable*" with the following description: "*Hello, I proposed this change to improve the test on MustacheParser. When a variable is improperly closed, a MustacheException is thrown.*".[9]

[7] https://github.com/addthis/stream-lib/pull/127/files
[8] https://github.com/addthis/stream-lib/pull/128/files
[9] https://github.com/spullara/mustache.java/pull/186/files

Listing 4.5: Test-improvement proposed to stream-lib developers with developers' suggestions.

```
@@ -52,4 +52,22 @@ public void testHash64ByteArrayOverload() {
assertEquals("MurmurHash.hash64(Object) given a byte[] did not match MurmurHash.
    hash64(String)",
hashOfString, MurmurHash.hash64(bytesAsObject));
}

+    // test the returned valued of hash functions against
+    // the reference implementation: https://github.com/aappleby/smhasher.git

+    @Test
+    public void testHash64() throws Exception {
+        final long actualHash = MurmurHash.hash64("hashthis");
+        final long expectedHash = -8896273065425798843L;
+
+        assertEquals(
+            "MurmurHash.hash64(String) returns wrong hash value",
+            expectedHash,
+            actualHash
+        );
+    }

+    @Test
+    public void testHash() throws Exception {
+        final long actualHash = MurmurHash.hash("hashthis");
+        final long expectedHash = -1974946086L;
+
+        assertEquals(
+            "MurmurHash.hash64(String) returns wrong hash value",
+            expectedHash,
+            actualHash
+        );
+    }
```

Listing 4.6: Test-improvement proposed to mustache.java developers.

```
@@ -63,4 +66,15 @@ public void testAbstractClassNoDots() throws IOException {
     mustache.execute(writer, scopes);
     writer.flush();
   }

+ @Test
+ public void testImproperlyClosedVariable() throws IOException {
+   try {
+     new DefaultMustacheFactory()
+         .compile(new StringReader("{{{#containers}} {{/containers}}"), "example");
+     fail("Should have throw MustacheException");
+   } catch (MustacheException actual) {
+     assertEquals(
+         "Improperly closed variable in example:1 @[example:1]",
+         actual.getMessage()
+     );
+   }
+ }
```

12 days later, a developer accepted the change, but noted that the test should be in another class. He closed the pull request and added the changes himself into the desired class.[10].

4.3.1.9 twilio-java

DSpot has been applied to amplify `RequestTest`. It identifies two new assertions that kill 4 more mutants. All killed mutants are between lines 260 and 265 in the method `equals()` of `Request`. The change specifies that an object `Request` is not equal to null nor an object of different type, *i.e.* `Object` here. The pull request was entitled "*add test equals() on request*", accompanied with the short description "*Hi, I propose this change to specify the equals() method of com.twilio.http.Request, against object and null value*". [11] Listing 4.7 shows this pull request.

A developer merged the change 4 days later.

4.3.1.10 jsoup

DSpot has been applied to amplify `AttributeTest`. It identifies one assertion that kills 13 more mutants. All mutants are in the method `hashcode` of Attribute. The pull request, shown in Listing 4.8, was entitled "*add test case for hashcode in attribute*" with

[10] the diff is same: https://github.com/spullara/mustache.java/commit/9efa19d595f893527ff218683e70db2ae4d8fb2d
[11] https://github.com/twilio/twilio-java/pull/334/files

Listing 4.7: Test-improvement proposed to twilio-java developers.

```
@@ -166,5 +166,13 @@ public void testRequiresAuthentication() {
+   assertTrue(request.requiresAuthentication());
+ }

+ @Test
+ public void testEquals() {
+   Request request = new Request(HttpMethod.DELETE, "/uri");
+   request.setAuth("username", "password");
+   assertFalse(request.equals(new Object()));
+   assertFalse(request.equals(null));
+ }
```

the following short description "*Hello, I propose this change to specify the hashCode of the object org.jsoup.nodes.Attribute.*"[12]:

Listing 4.8: Test-improvement proposed to jsoup developers.

```
@@ -17,4 +17,11 @@
 assertEquals(s + "=\"A" + s + "B\"", attr.html());
 assertEquals(attr.html(), attr.toString());
+ }

+ @Test
+ public void testHashCode() {
+   String s = new String(Character.toChars(135361));
+   Attribute attr = new Attribute(s, (("A" + s) + "B"));
+   assertEquals(111849895, attr.hashCode());
+ }
```

One developer highlighted the point that the `hashCode` method is an implementation detail, and it is not a relevant element of the API. Consequently, he did not accept our test improvement.

At this point, I have made two pull requests targeting `hashCode` methods. One accepted and one rejected. `hashCode` methods could require a different testing approach to validate the number of potential collisions in a collection of objects rather than checking or comparing the values of a few objects created for one explicit test case. The different responses obtained reflect the fact that developer teams and policies ultimately decide how to test the hash code protocol and the outcome could be different from different projects.

4.3.1.11 protostuff

DSpot has been applied to amplify `TailDelimiterTest`. It identifies a single assertion that kills 3 more mutants. All new mutants killed are in the method `writeTo` of `ProtostuffIOUtil` on lines 285 and 286, which is responsible to write a buffer into a

[12] https://github.com/jhy/jsoup/pull/840

given scheme. I proposed a pull request entitled “*assert the returned value of writeList*”, with the following short description “*Hi, I propose the following changes to specify the line 285-286 of io.protostuff.ProtostuffIOUtil.*”[13], shown in Listing 4.9.

Listing 4.9: Test-improvement proposed to protostuff developers.

```
@@ -144,7 +144,8 @@ public void testEmptyList() throws Exception
     ArrayList<Foo> foos = new ArrayList<Foo>();

     ByteArrayOutputStream out = new ByteArrayOutputStream();
-    writeListTo(out, foos, SerializableObjects.foo.cachedSchema());
+    final int bytesWritten =
+      writeListTo(out, foos, SerializableObjects.foo.cachedSchema()
+    );
+    assertEquals(0, bytesWritten);
     byte[] data = out.toByteArray();
```

A developer accepted the proposed changes the same day.

4.3.1.12 logback

DSpot has been applied to amplify `FileNamePattern`. It identifies a single assertion that kills 5 more mutant. Newly killed mutants were located at lines 94, 96 and 97 of the `equals` method of the `FileNamePattern` class. The proposed pull request was entitle “*test: add test on equals of FileNamePattern against null value*” with the following short description: “*Hello, I propose this change to specify the equals() method ofFileNamePattern against null value*”.[14]:

Listing 4.10: Test-improvement proposed to logback developers.

```
@@ -189,4 +190,11 @@ public void settingTimeZoneOptionHasAnEffect() {
     FileNamePattern fnp = new FileNamePattern("%d{hh, " + tz.getID() + "}",
         context);
     assertEquals(tz, fnp.getPrimaryDateTokenConverter().getTimeZone());
   }

+ @Test
+ public void testNotEqualsNull() {
+   FileNamePattern pp = new FileNamePattern("t", context);
+   assertFalse(pp.equals(null));
++ }
```

Even if the test asserts the contract that the `FileNamePattern` is not equals to null, and kills 5 more mutants, the lead developer does not get the point to test this behavior. The pull request has not been accepted.

[13] https://github.com/protostuff/protostuff/pull/212/files
[14] https://github.com/qos-ch/logback/pull/365/files

Table 4.4: Contributions of *A-Amplification* and *I-Amplification* on the amplified test method used to create a pull request.

Project	#*A-Amplification*	#*I-Amplification*
javapoet	2	2
mybatis-3	3	3
traccar	10	7
stream-lib	2	2
mustache	4	3
twilio	3	4
jsoup	34	0
protostuff	1	1
logback	2	2

4.3.1.13 retrofit

I did not manage to create a pull request based on the amplification of the test suite of retrofit. According to the result, the newly killed mutants are spread over all the code, and thus the amplified methods did not identify a missing contract specification. This could be explained by two facts: 1) the original test suite of retrofit is strong: there is no test class with low mutation score and a lot of them are very high mutation score, *i.e.* 90% and more; 2) the original test suite of retrofit uses complex test mechanism such as mock and fluent assertions of the form the `assertThat().isSomething()`. For the former point, it means that DSpot has been able to improve, even a bit, the mutation score of a very strong test suite, but not in targeted way that makes sense in a pull request. For the latter point, this puts in evidence the technical challenge of amplifying fluent assertions and mocking mechanisms.

4.3.1.14 Contributions of *A-Amplification* and *I-Amplification* to the Pull-requests

Table 4.4 summarizes the contribution of *A-Amplification* and *I-Amplification*, where a contribution means an source code modification added during the main amplification loop. In 8 cases over the 9 pull-requests, both *A-Amplification* and *I-Amplification* were necessary. Only the pull request on jsoup was found using only *A-Amplification*. This means that for all the other pull-requests, the new inputs were required to be able: 1) to kill new mutants and 2) to obtain amplified test methods that have values for the developers.

Note that this does not contradict with the fact that the pull requests are one-liners. Most one-liner pull requests contain both a new assertion and a new input. Consider the following Javapoet's one liner `assertFalse(x.equals(null))` (javapoet). In this example, although there is a single line starting with "assert", there is indeed a new input, the value "null".

RQ1: Would developers be ready to permanently accept improved test cases into the test repository?
Answer: 19 test improvements have been proposed to developers of notable open-source projects. 13/19 have been considered valuable and have been merged into the main test suite. The developers' feedback has confirmed the relevance, and also the challenges of automated test improvement.

In the area of automatic test improvement, this experiment is the first to put real developers in the loop, by asking them about the quality of automatically improved test cases. To the best of my knowledge, this is the first public report of automatically improved tests accepted by unbiased developers and merged in the master branch of open-source repositories.

4.3.2 Answer to RQ2

RQ2 To what extent are improved test methods considered as focused?

Table 4.5 presents the results for RQ2, RQ3 and RQ4.It is structured as follows. The first column is a numeric identifier that eases reference from the text. The second column is the name of test class to be amplified. The third column is the number of test methods in the original test class. The fourth column is the mutation score of the original test class. The fifth is the number of test methods generated by DSpot. The sixth is the number of amplified test methods that met the criteria explained in subsection 4.2.3. The seventh, eight and ninth are respectively the number of killed mutants of the original test class, the number of killed mutants of its amplified version and the absolute increase obtained with amplification, which is represented with a pictogram indicating the presence of improvement. The tenth and eleventh columns concern the number of killed mutants when only A-amplification is used. The twelfth is the time consumed by DSpot to amplify the considered test class. The upper part of the table is dedicated to test classes that have a high mutation score and the lower for the test classes that have low mutation score.

For RQ2, the considered results are in the sixth column of Table 4.5. The selection technique produces candidates that are focused in 25/26 test classes for which there are improved tests. For instance, considering test class TypeNameTest (#8), there are 19 improved test methods, and among them, 8 are focused per the definition and are worth considering to be integrated in the codebase. On the contrary, for test class ConcurrencyTest (#29), the technique cannot find any improved test method that matches the focus criteria presented in subsubsection 4.3.1.2. In this case, that improved test methods kill additional mutants in 27 different locations. Consequently, the intent of the new amplified tests can hardly be considered as clear.

Interestingly, for 4 test classes, even if there are more than one improved test methods,

Table 4.5: The effectiveness of test amplification with DSpot on 40 test classes: 22 well-tested (upper part) and 18 average-tested (lower part) real test classes from notable open-source Java projects.

ID	Class	# Orig. test methods	Mutation Score	# New test methods	Candidates for pull request	# Killed mutants orig.	# Killed mutants ampl.	Increase killed	# Killed mutants only A-ampl	Increase killed only A-ampl	Time (minutes)
	High mutation score										
1	TypeNameTest	12	50%	19	8	599	715	19% ↗	599	0.0% →	11.11
2	NameAllocatorTest	11	87%	0	0	79	79	0.0% →	79	0.0% →	4.76
3	MetaClassTest	7	58%	108	10	455	534	17% ↗	455	0.0% →	235.71
4	ParameterExpressionTest	14	91%	2	2	162	164	1% ↗	162	0.0% →	25.93
5	ObdDecoderTest	1	80%	9	2	51	54	5% ↗	51	0.0% →	2.20
6	MiscFormatterTest	1	72%	5	5	42	47	11% ↗	42	0.0% →	1.21
7	TestLookup3Hash	2	95%	0	0	464	464	0.0% →	464	0.0% →	6.76
8	TestDoublyLinkedList	7	92%	1	1	104	105	0.97% ↗	104	0.0% →	3.03
9	ArraysIndexesTest	1	53%	15	4	576	647	12% ↗	586	1% ↗	10.58
10	ClasspathResolverTest	10	67%	0	0	50	50	0.0% →	50	0.0% →	4.18
11	RequestTest	17	81%	4	3	141	156	10% ↗	141	0.0% →	60.55
12	PrefixedCollapsibleMapTest	4	96%	0	0	54	54	0.0% →	54	0.0% →	13.28
13	TokenQueueTest	6	69%	18	6	152	165	8% ↗	152	0.0% →	15.61
14	CharacterReaderTest	19	79%	71	9	309	336	8% ↗	309	0.0% →	57.06
15	TailDelimiterTest	10	71%	1	1	381	384	0.79% ↗	381	0.0% →	12.90
16	LinkBufferTest	3	48%	12	7	66	90	36% ↗	66	0.0% →	3.24
17	FileNamePatternTest	12	58%	27	9	573	686	19% ↗	573	0.0% →	25.08
18	SyslogAppenderBaseTest	1	95%	1	1	143	148	3% ↗	143	0.0% →	7.88
19	RequestBuilderAndroidTest	2	99%	0	0	513	513	0.0% →	513	0.0% →	0.04
20	CallAdapterTest	4	94%	0	0	55	55	0.0% →	55	0.0% →	7.30
39	ExecutorCallAdapterFactoryTest	7	62%	0	0	119	119	0.0% →	119	0.0% →	0.09
40	CallTest	35	69%	3	1	642	644	0.32% ↗	642	0.0% →	52.84
	Low mutation score										
21	FieldSpecTest	2	31%	12	4	223	316	41% ↗	223	0.0% →	4.44
22	ParameterSpecTest	2	32%	11	5	214	293	36% ↗	214	0.0% →	3.66
23	WrongNamespacesTest	2	8%	6	1	78	249	219% ↗	249	219% ↗	29.70
24	WrongMapperTest	1	8%	3	1	97	325	235% ↗	325	235% ↗	7.13
25	ProgressProtocolDecoderTest	1	16%	2	1	18	27	50% ↗	23	27% ↗	1.30
26	IgnitionEventHandlerTest	1	22%	0	0	13	13	0.0% →	13	0.0% →	0.77
27	TestICardinality	2	7%	0	0	19	19	0.0% →	19	0.0% →	2.13
28	TestMurmurHash	2	17%	40	2	52	275	428% ↗	174	234% ↗	2.18
29	ConcurrencyTest	2	28%	2	0	210	342	62% ↗	210	0.0% →	315.56
30	AbstractClassTest	2	34%	28	4	383	475	24% ↗	405	5% ↗	12.67
31	AllTimeTest	3	42%	0	0	163	163	0.0% →	163	0.0% →	0.02
32	DailyTest	3	42%	0	0	163	163	0.0% →	163	0.0% →	0.02
33	AttributeTest	2	36%	33	11	178	225	26% ↗	180	1% ↗	10.76
34	AttributesTest	5	52%	9	6	316	322	1% ↗	316	0.0% →	6.21
35	CodedDataInputTest	1	1%	0	0	5	5	0.0% →	5	0.0% →	3.58
36	CodedInputTest	1	27%	29	28	108	166	53% ↗	108	0.0% →	0.88
37	FileAppenderResilience_AS_ROOT_Test	1	4%	0	0	4	4	0.0% →	4	0.0% →	0.65
38	Basic	1	10%	0	0	6	6	0.0% →	6	0.0% →	0.89

the selection technique only returns one focus candidate (#23, #24, #25, #40). In those cases, there are two possible different reasons: 1) there are several focused improved tests, yet they all specify the same application method (this is the case for #40) 2) there is only one improved test method that is focused (this is the case for #23, #24, and #25)

To conclude, according to this benchmark, DSpot proposes at least one and focused improved test in all but one cases. From the developer viewpoint, DSpot is not overwhelming it proposes a small set of suggested test changes, which are ordered, so that even with a small time budget to improve the tests, the developer is pointed to the most interesting case.

> *RQ2: To what extent are improved test methods considered as focused?*
> Answer: In 25/26 cases, the improvement is successful at producing at least one focused test method, which is important to save valuable developer time in analyzing the suggested test improvements.

4.3.3 Answer to RQ3

RQ3: To what extent do improved test classed kill more mutants than developer-written test classes?

In 26 out of 40 cases, DSpot is able to amplify existing test cases and improves the mutation score (MS) of the original test class.

For example, let us consider the first row, corresponding to `TypeNameTest`. This test class originally includes 12 test methods that kill 599 mutants. The improved, amplified version of this test class kills 715 mutants, *i.e.* 116 new mutants are killed. This corresponds to an increase of 19% in the number of killed mutants.

First, let's discuss the amplification of the test classes that can be considered as being already good tests since they originally have a high mutation score: those good test classes are the 24 tests in Table 4.5. There is a positive increase of killed mutants for 17 cases. This means that even when human developers write good test cases, DSpot is able to improve the quality of these test cases by increasing the number of mutants killed. In addition, in 15 cases, when the amplified tests kill more mutants, this goes along with an increase of the number of expressions covered with respect to the original test class.

For those 24 good test classes, the increase in killed mutants varies from 0,3%, up to 53%. A remarkable aspect of these results is that DSpot is able to improve test classes that are initially extremely strong, with an original mutation score of 92% (ID:8) or even 99%

(ID:20 and ID:21). The improvements in these cases clearly come from the double capacity of DSpot at exploring more behaviors than the original test classes and at synthesizing new assertions.

Still looking to the upper part of Table 4.5 (the good test classes), focus now on the relative increase in killed mutants (column "Increase killed"). The two extreme cases are `CallTest` (ID:24) with a small increase of 0.3% and `CodeInputTest` (ID:18) with an increase of 53%. `CallTest` (ID:24) initially includes 35 test methods that kill 69% of 920 covered mutants. Here, DSpot runs for 53 minutes and succeeds in generating only 3 new test cases that kill 2 more mutants than the original test class, and the increase in mutation score is only minimal. The reason is that input amplification does not trigger any new behavior and assertion amplification fails to observe new parts of the program state. Meanwhile, DSpot succeeds in increasing the number of mutants killed by `CodeInputTest` (ID:18) by 53%. Considering that the original test class is very strong, with an initial mutation score of 60%, this is a very good achievement for test amplification. In this case, the *I-Amplification* applied easily finds new behaviors based on the original test code. It is also important to notice that the amplification and the improvement of the test class goes very fast here (only 52 seconds).

One can notice 4 cases (IDs:3, 13, 15, 24) where the number of new test cases is greater than the number of newly killed mutants. This happens because DSpot amplifies test cases with different operators in parallel. While DSpot keeps only amplified test methods that kill new mutants, it happens that the same mutant is newly killed by two different amplified tests generated in parallel threads.
In this case, DSpot keeps both amplified test methods.

There are 7 cases with high mutation score for which DSpot does not improve the number of killed mutants. In 5 of these cases, the original mutation score is greater than 87% (IDs: 2, 7, 12, 21, 22), and DSpot does not manage to synthesize improved inputs to cover new mutants and eventually kill them. In some cases DSpot cannot improve the test class because they rely on an external resource (a jar file), or use utility methods that are not considered as test methods by DSpot and hence are not modified by our tool.

Now consider the tests in the lower part of Table 4.5. Those tests are weaker because they have a lower mutation score. When amplifying weak test classes, DSpot improves the number of killed mutants in 9 out of 16 cases. On a per test class basis, this does not differ much from the good test classes. However, there is a major difference when one considers the increase itself: the increases in number of killed mutants range from 24% to 428%. Also, one can observe a very strong distinction between test classes that are greatly improved and test classes that are not improved at all (9 test classes are much improved, 7 test classes cannot be improved at all, the increase is 0%). In the former case, test classes provide a good seed for amplification. In the latter case, there are test classes that are designed in a way that prevents amplification because they use external processes, or depend on administration permission, shell commands and external data sources; or extensively use mocks or factories; or simply very small test methods that do not provide a good potential to DSpot to perform effective amplification.

> *RQ3: To what extent do improved test classes kill more mutants than manual test classes?*
> Answer: In this quantitative experiment on automatic test improvement, DSpot significantly improves the capacity of test classes at killing mutants in 26 out 40 of test classes, even in cases where the original test class is already very strong. Automatic test improvement works particularly well for weakly tested classes (lower part of Table 4.5): the mutation score of three classes is increased by more than 200%.

The most notable point of this experiment is that there are considered tests that are already really strong (Table 4.5), with mutation score in average of 78%, with the surprising case of a test class with 99% mutation score that DSpot is able to improve.

4.3.4 Answer to RQ4

What is the contribution of *I-Amplification* and *A-Amplification* to the effectiveness of automated test improvement?

The relevant results are reported in the tenth and eleventh column of Table 4.5. They give the number of killed mutants and the relative increase of the number of killed mutants when only using *A-Amplification*.

For instance, for `TypeNameTest` (first row, id #1), using only *A-Amplification* kills 599 mutants, which is exactly the same number of the original test class. In this case, both the absolute and relative increase are obviously zero. On the contrary, for `WrongNamespacesTest` (id #27), using only *A-Amplification* is very effective, it enables DSpot to kill 249 mutants, which, compared to the 78 originally killed mutants, represents an improvement of 219%.

Now, when aggregating over all test classes, the results indicate that *A-Amplification* only is able to increase the number of mutants killed in 7 / 40 test classes. Increments range from 0.31% to 13%. Recall that when DSpot runs both *I-Amplification* and *A-Amplification*, it increases the number of mutants killed in 26 / 40 test classes, which shows that it is indeed the combination of *A-Amplification* and *I-Amplification* which is effective.

Note that *A-Amplification* performs as well as *I-Amplification* + *A-Amplification* in only 2/40 cases (ID:27 and ID:28). In this case, all the improvement comes from the addition of new assertions, and this improvement is dramatic (relative increase of 219% and 235%).

The limited impact of *A-Amplification* alone has several causes. First, many assertions in the original test cases are already good and precisely specify the expected behavior for the test case. Second, it might be due to the limited observability of the program under test (*i.e.*, there is a limited number of points where assertions over the program state can be expressed). Third, it happens when one test case covers global properties across many methods: test #28 `WrongMapperTest` specifies global properties, but is not well suited to observe fine grained behavior with additional assertions. This latter case is common among the weak test classes of the lower part of Table 4.5.

> *RQ4: What is the contribution of* I-Amplification *and* A-Amplification *to the effectiveness of test amplification?*
> Answer: The conjunct run of *I-Amplification* and *A-Amplification* is the best strategy for DSpot to improve manually-written test classes. This experiment has shown that *A-Amplification* is ineffective, in particular on tests that are already strong.

To the best of my knowledge, this experiment is the first to evaluate the relative contribution of *I-Amplification* and *A-Amplification* to the effectiveness of automatic test improvement.

4.4 Threats to Validity

RQ1 The major threat to RQ1 is that there is a potential bias in the acceptance of the proposed pull requests. For instance, if I propose pull requests to colleagues, they are more likely to merge them. However, this is not the case here. In this evaluation, I am unknown to all considered projects. The developers who study the DSpot pull requests are independent from our group and social network. Since I was unknown for the pull request reviewer, this is not a specific bias towards acceptance or rejection of the pull request.

RQ2 The technique used to select focused candidates is based on the proportion of mutant killed and the absolute number of modification done by the amplification. However, it may happen that some improvements that are not focused per our definition would still be considered as valuable by developers. Having such false negative is a potential threat to validity.

RQ3 A threat to RQ3 relates to external validity: if the considered projects and tests are written by amateurs, the findings would not hold for serious software projects. However, the experimentation only considers real-world applications, maintained by professional and esteemed open-source developers. This means that considered tests are arguably among the best of the open-source world, aiming at as strong construct validity as possible.

RQ4. The main threat to RQ4 relates to internal validity: since the results are of computational nature, a bug in the implementation or experimental scripts may threaten the findings. All the code is publicly-available for other researchers to reproduce the experiment and spot the bugs, if any.

Oracle. DSpot generates new assertions based on the current behavior of the program. If the program contains a bug, the resulting amplified test methods would enforce this bug. This is an inherent threat, inherited from [Xie 2006], which is unavoidable when no additional oracle is available, but only the current version of the program. To that extent, the best usage of DSpot is to improve the test suite of a supposedly almost correct version of the program.

4.5 Conclusion

This first evaluation of DSpot gave 2 main results:

1) 19 amplified test methods have been proposed to be integrated in test suites from open-source projects. The developers of these projects reviewed amplified test methods, proposed in the form of pull requests. 14 of them have been merged permanently in the test suite of projects. It means that developers value amplified test methods produced by DSpot. It also means that amplified test methods, obtained using DSpot, are increasing the developers' confidence in the correctness of their program;

2) 40 test classes have been amplified to improve their mutation score. 26 of them result with an actual improvement of the mutation score. This shows that DSpot is able to improve existing test suites.

In this chapter, the mutation score has been used to amplified test methods. The mutation score is a measure of the test suites quality to detect small behavioral changes, as mutants emulate them. However, the behavioral changes introduces by mutants may be different from those in real commits. That is to say, it does not show any evidence that DSpot would be able to detect behavioral change introduced by commits, which are typically larger, more complex and significant.

In the next chapter, I investigate the capacity of DSpot to improve existing test methods in order to detect real behavioral changes, introduced by commits. To do so, I confront DSpot to real modifications done by developers on their code base from GitHub. In addition to this, the next chapter exposes an enhancement of DSpot's usage and puts it in the continuous integration.

CHAPTER 5

Test Amplification For Behavioral Changes Detection Of Commits

Contents

In collaborative software projects, developers work in parallel on the same code base. Every time a developer integrates her changes, she submits them in the form of a *commit* to a version control system. The *Continuous Integration* (CI) server [Fowler 2006] merges the commit with the master branch, compiles and automatically runs the test suite to check that the commit behaves as expected. Its ability to detect bugs early makes CI an essential contribution to quality assurance [Hilton 2016, Duvall 2007]. However, the effectiveness of Continuous Integration depends on one key property: each commit should include at least one test case t_{new} that specifies the intended change.

For instance, assume one wants to integrate a bug fix. In this case, the developer is expected to include a new test method, t_{new}, that specifies the program's desired behavior after the bug fix is applied. This can be mechanically verified: t_{new} should fail on the version of the code that does not include the fix (the *pre-commit* version), and pass on the version that includes the fix (the *post-commit* version). However, many commits either do not include a t_{new} or t_{new} does not meet this fail/pass criterion. The reason is that developers sometimes cut corners because of lack of time, expertise or discipline. This is the problem addressed in this chapter.

In this chapter, I detail an extension of DSpot, called DCI (DSpot-CI), and its evaluation. The goal is to automatically generate test methods for each commit that is submitted to the CI. In particular, to generate a test method t_{gen} that specifies the behavioral change of each commit.

DCI works in two steps: First, it analyzes the test methods of the pre-commit version and selects the ones that exercise the parts of the code modified by the commit. Second, it applies DSpot on this subset of test methods. The test selection is done only on amplified test methods that are relevant, *i.e.* t_{gen} *passes* on the pre-commit version and *fails* on the post-commit version.

This evaluation has been performed on 60 commits from 6 open-source projects on GitHub. The result is that DCI has been able to obtain amplified test methods detecting 25 behavioral changes.

To sum up, the contributions of this chapter are:

- DCI (**D**spot-**CI**), a complete approach to obtain automatically test methods that detect behavioral changes.
- An open-source implementation of DCI for Java.
- A curated benchmark of 60 commits that introduce a behavioral change and include a test case to detect it, selected from 6 notable open source Java projects[1].
- A comprehensive evaluation based on 4 research questions that combines quantitative and qualitative analysis with manual assessment.

Note that this chapter is a to be published article [Danglot 2019a]. The remainder of this chapter is as follows: Section 5.1 motivates this chapter and gives the background. Section 5.2 exposes the technical extension of DSpot: an approach for commit-based test selection. Section 5.3 introduces our benchmark of commits, the evaluation protocol and

[1] https://github.com/STAMP-project/dspot-experiments.git

the results of our experiments on 50 real commits. Section 5.5 relates the threats validity and actions that have been taken to overcome them. and Section 5.6 concludes this chapter.

5.1 Motivation & Background

In this section, I introduce an example to motivate the need to generate new tests that specifically target the behavioral change introduced by a commit. Then I introduce the key concepts on which the solution has been elaborated to address this challenging test generation task.

5.1.1 Motivating Example

On August 10, a developer pushed a commit to the master branch of the XWiki-commons project. The change[2], displayed in Figure 5.1, adds a comparison to ensure the equality of the objects returned by `getVersion()`. The developer did not write a test method nor modify an existing one.

Figure 5.1: Commit 7e79f77 on XWiki-Commons that changes the behavior without a test.

```
&& Objects.equals(getDataFormat(), ((FilterStreamType) object).getDataFormat());
&& Objects.equals(getDataFormat(), ((FilterStreamType) object).getDataFormat())
&& Objects.equals(getVersion(), ((FilterStreamType) object).getVersion());
```

In this commit, the intent is to take into account the `version` (from method `getVersion`) in the `equals` method. This change impacts the behavior of all methods that use it, the method `equals` being a highly used. Such a central behavioral change may impact the whole program, and the lack of a test method for this new behavior may have dramatic consequences in the future. Without a test method, this change could be reverted and go undetected by the test suite and the Continuous Integration server, *i.e.* the build would still pass. Yet, a user of this program would encounter new errors, because of the changed behavior. The developer took a risk when committing this change without a test case.

DCI aims at mitigating such risk: ensuring that every commit include a new test method or a modification of an existing test method. In this chapter, I study DSpot's ability to automatically obtain a test method that highlights the behavioral change introduced by a commit. This test method allows to identify the behavioral difference between the two versions of the program. The goal is to use this new test method to ensure that any changed behavior can be caught in the future.

Following, the vision of DCI's usage: when Continuous Integration is triggered, rather than just executing the test suite to find regressions, it could also run an analysis of the

[2] `https://github.com/xwiki/xwiki-commons/commit/7e79f77`

commit to know if it contains a behavioral change, in the form of a new method or the modification of an existing one. If there is no appropriate test method to detect a behavioral change, the approach would provide one. DCI would take as input the commit and a test suite, and generate a new test method that detects the behavioral change.

5.1.2 Practicability

Following, the vision of an integration scenario of DCI:

A developer commits a change into the program. The Continuous Integration service is triggered; the CI analyzes the commit. There are two potential outcomes: 1) the developer provided a new test method or a modification to an existing one. In this case, the CI runs as usual, *e.g.* it executes the test suite; 2) the developer did not provide a new test nor the modification of an existing one, the CI runs DCI on the commit to obtain a test method that detects the behavioral change and present it to the developer. The developer can then validate the new test method that detects the behavioral change. Following the test selection, the new test method passes on the pre-commit version but fails on the post-commit version. The current amplified test method cannot be added to the test suite, since it fails. However, this test method is still useful, since one has only to negate the failing assertions, *e.g.* change an `assertTrue` into an `assertFalse`, to obtain a valid and passing test method that explicitly executes the new behavior. This can be done manually or automatically with approaches such as ReAssert[Daniel 2009a].

DCI has been designed to be easy to use. The only cost of DCI is the time to set it up: in the ideal, happy-path case, it is meant to be a single command line through Maven goals. Once DCI is set up in continuous integration, it automatically runs at each commit and developers directly benefit from amplified test methods that strengthen the existing test suite.

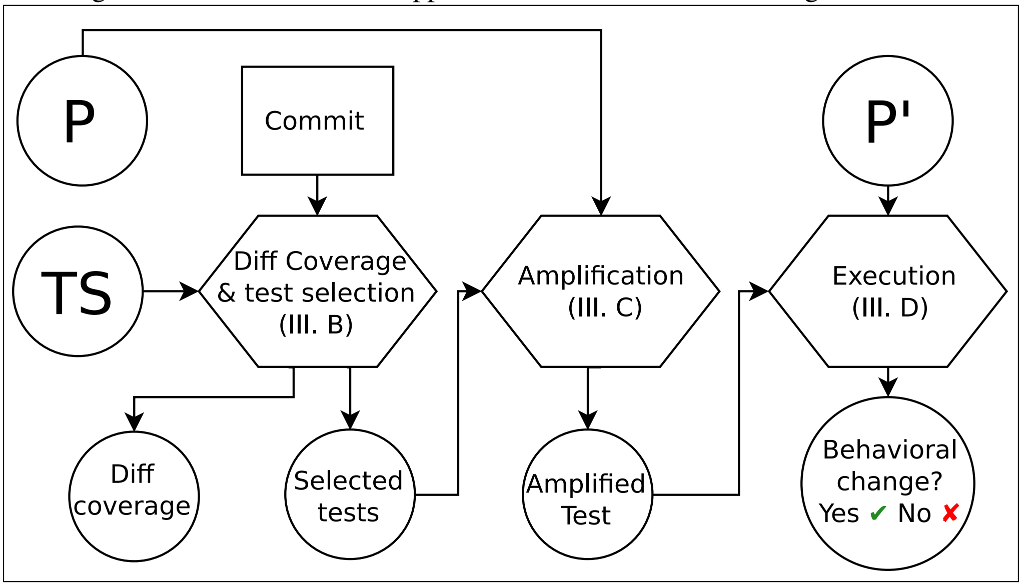
5.1.3 Behavioral Change

A *behavioral change* is a source-code modification that triggers a new state for some inputs [Saff 2004]. Considering the pre-commit version P and the post-commit version P' of a program, the commit introduces a behavioral change if it is possible to implement a test method that can trigger and observe the change, *i.e.*, it passes on P and fails on P', or the opposite. In short, the behavioral change must have an impact on the observable behavior of the program.

5.1.4 Behavioral Change Detection

Behavioral change detection is the task of identifying a behavioral change between two versions of the same program. In this chapter, I propose a novel approach to detect behav-

Figure 5.2: Overview of the approach to detect behavioral changes in commits.



ioral changes based on test amplification.

5.2 Behavioral Change Detection Approach

5.2.1 Overview of DCI

DCI takes as input a program, its test suite, and a commit modifying the program. The commit, as done in version control systems, is basically the diff between two consecutive versions of the program.

DCI outputs new test methods that detect the behavioral difference between the pre- and post-commit versions of the program. The new tests pass on a given version, but fail on the other, demonstrating the presence of a behavioral change captured.

DCI computes the code coverage of the diff and selects test methods accordingly. Then it applies DSpot to amplify selected test methods. The resulting amplified test methods detect the behavioral change.

Figure 5.2 sums up the different phases of the approach: 1) Compute the diff coverage and select the tests to be amplified; 2) Amplify the selected tests based on the pre-commit version; 3) Execute amplified test methods against the post-commit version, and keep the failing test methods. This process produces test methods that pass on the pre-commit version, fail on the post-commit version, hence they detect at least one behavioral change introduced by a given commit.

5.2.2 Test Selection and Diff Coverage

DCI implements a feature that: 1. reports the diff coverage of a commit, and 2. selects the set of unit tests that execute the diff. To do so, DCI first computes the code coverage for the whole test suite. Second, it identifies the test methods that hit the statements modified by the diff. Third, it produces the two outcomes elicited earlier: the diff coverage, computed as the ratio of statements in the diff covered by the test suite over the total number of statements in the diff and the list of test methods that cover the diff. Then, it selects only test methods that are present in pre-commit version (*i.e.*, it ignores the test methods added in the commit, if any). The final list of test methods that cover the diff is then used to seed the amplification process.

5.2.3 Test Amplification

Once DCI have the initial tests that cover the diff, DCI amplifies them using DSpot. Since DCI uses DSpot, DCI have also two mode: 1)DCI-*A-Amplification* that uses only *A-Amplification* and 2)DCI-*I-Amplification* that uses both *A-Amplification* and *I-Amplification*.

5.2.4 Execution and Change Detection

The final step performed by DCI consists in checking whether that the amplified test methods detect behavioral changes. Because DCI amplifies test methods using the pre-commit version, all amplified test methods pass on this version, by construction. Consequently, for the last step, DCI runs the amplified test methods only on the post-commit version. Every test that fails is in fact detecting a behavioral change introduced by the commit, and is a success. DCI keeps the tests that successfully detect behavioral changes. Note that if the amplified test method is not executable on the post-commit version, *e.g.* the API has been modified, the amplified test method is discarded.

5.2.5 Implementation

DCI is implemented in Java and is built on top of the OpenClover and Gumtree [Falleri 2014] libraries. It computes the global coverage of the test suite with OpenClover, which instruments and executes the test suite. Then, it uses Gumtree to have an AST representation of the diff. DCI matches the diff with the test that executes those lines. Through its Maven plugin, DCI can be seamlessly implemented into continuous integration. DCI is publicly available on GitHub.[3]

[3] https://github.com/STAMP-project/dspot/tree/master/dspot-diff-test-selection

5.3 Evaluation

The evaluation of DCI relies on 4 research questions:

5.3.1 Research Questions

RQ1: To what extent are DCI-*A-Amplification* and DCI-*I-Amplification* able to produce amplified test methods that detect the behavioral changes?
RQ2: What is the impact of the number of iteration performed by DCI-*I-Amplification* ?
RQ3: What is the effectiveness of our test selection method?
RQ4: How do human and generated tests that detect behavioral changes differ?

5.3.2 Benchmark

To the best of my knowledge, there is no benchmark of commits in Java with behavioral changes in the literature. Consequently, I devise a project and commit selection procedure in order to construct a benchmark for the evaluation.

Project selection The evaluation needs software projects that are

1) publicly-available;
2) written in Java;
3) and use continuous integration.

The projects has been selected from the dataset in [Vera-Pérez 2018b] and [Danglot 2019c], which is composed of mature Java projects from GitHub.

Commit selection Commits has been taken in inverse chronological order, from newest to oldest. I select the first ten commits that match the following criteria:

1) the commit modifies Java files (most behavioral changes are source code changes. It is known that behavioral changes can be introduced in other ways, such as modifying dependencies or configuration files [Hilton 2018b]. However, such modifications are not the target of DCI.

2) the commit provides or modifies a manually written test that detects a behavioral change. To verify this property, I execute the test on the pre-commit version. If it fails, it means that the test detects at least 1 behavioral change. This test will be used as a *ground-truth test* in **RQ4**.

3) the changes of the commit must be covered by the pre-commit test suite. To do so, I compute the diff coverage. If the coverage is 0%, the commit is discarded. This is done because if the change is not covered, any test methods cannot be selected to be amplified, which is what a part of the evaluation.

Together, these criteria ensure that all selected commits:

1) introduce behavioral changes;

2) at least one test method can be used as ground-truth since it detects a behavioral change;

3) at least one test method executes the diff and can be used to seed the amplification process;

4) there is no structural change in the commit between both versions, *e.g.* no change in method signature and deletion of classes (this is ensured since the pre-commit test suite compiles and runs against the post-commit version of the program and vice-versa).

Table 5.1: Selected projects and commits dataset.

project	LOC	start date	end date	#total commits	#discarded commits	#matching commits	#selected commits
commons-io	59607	9/10/2015	9/29/2018	385	375	16(4.16%)	10
commons-lang	77410	11/22/2017	10/9/2018	227	217	13(5.73%)	10
gson	49766	6/14/2016	10/9/2018	159	149	13(8.18%)	10
jsoup	20088	12/21/2017	10/10/2018	50	40	11(22.00%)	10
mustache.java	10289	7/6/2016	04/18/2019	68	58	11(16.18%)	10
xwiki-commons	87289	10/31/2017	9/29/2018	687	677	23(3.35%)	10
summary	304449	9/10/2015	04/18/2019	avg(262.67)	avg(252.67)	avg(14.50(9.93%))	60

Final benchmark Table 5.1 shows the main descriptive statistics on the benchmark dataset. The first column is the name of the considered project; The second column is the number of lines of code computed with cloc; The third column is the date of the oldest commit for the project; The fourth column is the date of the newest commit for the project; The fifth, sixth and seventh are respectively the total number of commit we analyze, the total number of commits discarded, the number of commits that match all the inclusion criteria but the third (there is no test in the pre-commit that execute the change), and the number of commit selected. The last row reports a summary of the benchmark with the total number of lines of code, the oldest and the newest dates, the average number of commits analyzed, the average number of commits discarded, the average number of commits matching all the criteria but the third. Note that the benchmark is only composed of recent commits from notable open-source projects and is available on GitHub at `https://github.com/STAMP-project/dspot-experiments`.

5.3.3 Protocol

To answer **RQ1**, DCI-*A-Amplification* and DCI-*I-Amplification* is executed on the benchmark projects. The total number of behavioral changes successfully detected by DCI is reported. That is to say the number of commits for which DCI generates at least 1 test method that passes on the pre-commit version but fails on the post-commit version. Also, 1 case study of a successful behavioral change detection is discussed.

To answer **RQ2**, DCI-*I-Amplification* with 1, 2 and 3 iterations is executed on the benchmark projects. The number of behavioral changes successfully detected for each number of iterations in the main loop is reported. Also, the number amplified test methods that detect the behavioral changes for each commit for 10 different seeds to study the impact of the randomness on the output of DSpot is reported. A Kruskal-Wallis test statistic is performed on these numbers.

To answer **RQ2**, we run DCI-*I-Amplification* for 1, 2 and 3 iterations on the benchmark projects. The number of behavioral changes successfully detected for each number of iterations in the main loop is reported. I analyze the number of amplified test methods that detect the behavioral changes for each commit for 10 different seeds in addition to the reference run with default seed, totalling 11 runs. The objective here is to study the impact of the randomness on the output of DCI-*I-Amplification* using 1 iteration. Since each experiment takes very long to run, I choose to use only 1 iteration for each seed. Doing these 10 different executions using 3 iterations would result with an execution that would last almost 1000 hours of cpu-time, which is infeasible. I compute the confidence interval on the number of successes, *i.e.* the number of time DCI generates at least one amplified test method that detects the behavioral change, in order to measure the uncertainty of the result. To do this, I use Python libraries *scipy* and *numpy*, and consider a confidence level of 95%. Per this open-science approach, the interested reader has access to both the raw data and the script computing the confidence interval. [4]

For **RQ3**, the test selection method is considered effective if the tests selected to be amplified semantically relate to the code changed by the commit. To assess this, 1 commit per project in the benchmark is selected. Then the automatically selected tests for this commit is manually analyzed to tell whether there are semantically related to the behavioral changes in the commit.

To answer **RQ4**, the ground-truth tests written or modified by developers in the selected commits is used. This ground-truth test method is compared to the amplified test methods that detect behavioral changes, for 1 commit per project.

5.3.4 Results

The overall results are reported in Table 5.2. This table can be read as follow: the first column is the name of the project; the second column is the shortened commit id; the third column is the commit date; the fourth column column is the total number of test methods executed when building that version of the project; the fifth and sixth columns are

[4] `https://github.com/STAMP-project/dspot-experiments/tree/master/src/main/python/april-2019`

Table 5.2: Evaluation of DCI on 60 commits from 6 large open-source projects.

	id	date	#Test	#Modified Tests	+ / -	Cov	#Selected Tests	#AAMPL Tests	Time	#SBAMPL Tests	Time
commons-io	c6b8a38	6/12/18	1348	2	104 / 3	100.0	3	0	10.0s	0	98.0s
	2736b6f	12/21/17	1343	2	164 / 1	1.79	8	0	19.0s	✓ (12)	76.3m
	a4705cc	4/29/18	1328	1	37 / 0	100.0	2	0	10.0s	0	38.1m
	f00d97a	5/2/17	1316	10	244 / 25	100.0	2	✓ (1)	10.0s	✓ (39)	27.0s
	3378280	4/25/17	1309	2	5 / 5	100.0	1	✓ (1)	9.0s	✓ (11)	24.0s
	703228a	12/2/16	1309	1	6 / 0	50.0	8	0	19.0s	0	71.0m
	a7bd568	9/24/16	1163	1	91 / 83	50.0	8	0	20.0s	0	65.2m
	81210eb	6/2/16	1160	1	10 / 2	100.0	1	0	8.0s	✓ (8)	23.0s
	57f493a	11/19/15	1153	1	15 / 1	100.0	8	0	7.0s	0	54.0s
	5d072ef	9/10/15	1125	12	74 / 34	68.42	25	✓ (6)	29.0s	✓ (1538)	2.2h
	total						66	8	2.4m	1608	6.5h
	average						6.60	0.80	14.5s	160.80	38.8m
commons-lang	f56931c	7/2/18	4105	1	30 / 4	25.0	42	0	2.4m	0	8.5m
	87937b2	5/22/18	4101	1	114 / 0	77.78	16	0	35.0s	0	18.1m
	09ef69c	5/18/18	4100	1	10 / 1	100.0	4	0	16.0s	0	98.8m
	3fadfdd	5/10/18	4089	1	7 / 1	100.0	9	0	17.0s	✓ (4)	17.2m
	e7d16c2	5/9/18	4088	1	13 / 1	33.33	7	0	16.0s	✓ (2)	15.1m
	50ce8c4	3/8/18	4084	4	40 / 1	90.91	2	✓ (1)	28.0s	✓ (135)	2.0m
	2e9f3a8	2/11/18	4084	2	79 / 4	30.0	47	0	79.0s	0	66.5m
	c8e61af	2/10/18	4082	1	8 / 1	100.0	10	0	17.0s	0	16.0s
	d8ec011	11/12/17	4074	1	11 / 1	100.0	5	0	31.0s	0	2.3m
	7d061e3	11/22/17	4073	1	16 / 1	100.0	8	0	17.0s	0	11.4m
	total						150	1	6.7m	141	4.0h
	average						15.00	0.10	40.5s	14.10	24.0m
gson	b1fb9ca	9/22/17	1035	1	23 / 0	50.0	166	0	4.2m	0	92.5m
	7a9fd59	9/18/17	1033	2	21 / 2	83.33	14	0	15.0s	✓ (108)	2.1m
	03a72e7	8/1/17	1031	2	43 / 11	68.75	371	0	7.7m	0	3.2h
	74e3711	6/20/17	1029	1	68 / 5	8.0	1	0	4.0s	0	16.0s
	ada597e	5/31/17	1029	2	28 / 3	100.0	5	0	8.0s	0	8.7m
	a300148	5/31/17	1027	7	103 / 2	18.18	665	0	9.2m	✓ (6)	4.9h
	9a24219	4/19/17	1019	1	13 / 1	100.0	36	0	2.2m	0	48.9m
	9e6f2ba	2/16/17	1018	2	56 / 2	50.0	9	0	32.0s	✓ (2)	8.5m
	44cad04	11/26/16	1015	1	6 / 0	100.0	2	0	15.0s	✓ (37)	40.0s
	b2c00a3	6/14/16	1012	4	242 / 29	60.71	383	0	7.9m	0	3.6h
	total						1652	0	32.4m	153	14.4h
	average						165.20	0.00	3.2m	15.30	86.5m
jsoup	426ffe7	5/11/18	668	4	27 / 46	64.71	27	✓ (2)	42.0s	✓ (198)	33.6m
	a810d2e	4/29/18	666	1	27 / 1	80.0	5	0	10.0s	0	26.6m
	6be19a6	4/29/18	664	1	23 / 1	50.0	50	0	69.0s	0	67.7m
	e38dfd4	4/28/18	659	1	66 / 15	90.0	18	0	35.0s	0	12.5m
	e9feec9	4/15/18	654	1	15 / 3	100.0	4	0	9.0s	0	95.0s
	0f7e0cc	4/14/18	653	2	56 / 15	84.62	330	0	6.5m	✓ (36)	11.8h
	2c4e79b	4/14/18	650	2	82 / 2	50.0	44	0	67.0s	0	4.7h
	e5210d1	12/22/17	647	1	3 / 3	100.0	14	0	9.0s	0	4.9m
	df272b7	12/22/17	647	2	17 / 1	100.0	13	0	9.0s	0	4.6m
	3676b13	12/21/17	648	6	104 / 12	38.46	239	0	6.2m	✓ (52)	6.8h
	total						744	2	16.8m	286	25.8h
	average						74.40	0.20	101.0s	28.60	2.6h
mustache.java	a1197f7	1/25/18	228	1	43 / 57	77.78	131	0	11.8m	✓ (204)	10.1h
	8877027	11/19/17	227	1	22 / 2	33.33	47	0	7.3m	0	100.2m
	d8936b4	2/1/17	219	2	46 / 6	60.0	168	0	12.7m	0	84.2m
	88718bc	1/25/17	216	2	29 / 1	100.0	1	✓ (1)	7.0s	✓ (149)	3.7m
	339161f	9/23/16	214	2	32 / 10	77.78	123	0	8.6m	✓ (1312)	5.8h
	774ae7a	8/10/16	214	2	17 / 2	100.0	11	0	66.0s	✓ (124)	6.8m
	94847cc	7/29/16	214	2	17 / 2	100.0	95	0	11.5m	✓ (2509)	21.4h
	eca08ca	7/14/16	212	4	47 / 10	80.0	18	0	87.0s	0	41.8m
	6d7225c	7/7/16	212	2	42 / 4	80.0	18	0	87.0s	0	40.1m
	8ac71b7	7/6/16	210	10	167 / 31	40.0	20	0	2.1m	✓ (124)	5.6m
	total						632	1	58.1m	4422	42.0h
	average						63.20	0.10	5.8m	442.20	4.2h
xwiki-commons	ffc3997	7/27/18	1081	0	125 / 18	21.05	1	0	29.0s	0	18.0s
	ced2635	8/13/18	1081	1	21 / 14	60.0	5	0	93.0s	0	2.5h
	10841b1	8/1/18	1061	1	107 / 19	30.0	51	0	5.7m	0	3.4h
	848c984	7/6/18	1074	1	154 / 111	17.65	1	0	28.0s	0	18.0s
	adfefec	6/27/18	1073	1	17 / 14	40.0	22	✓ (1)	76.0s	✓ (3)	14.9m
	d3101ae	1/18/18	1062	2	71 / 9	20.0	4	✓ (1)	72.0s	✓ (31)	41.4m
	a0e8b77	1/18/18	1062	2	51 / 8	42.86	4	✓ (1)	72.0s	✓ (60)	42.1m
	78ff099	12/19/17	1061	1	16 / 0	33.33	2	0	68.0s	✓ (4)	6.6m
	1b79714	11/13/17	1060	1	20 / 5	60.0	22	0	78.0s	0	17.9m
	6dc9059	10/31/17	1060	1	4 / 14	88.89	22	0	79.0s	0	20.5m
	total						134	3	15.7m	98	8.2h
	average						13.40	0.30	94.3s	9.80	49.5m
	total						3378	9(15)	2.2h	25(6708)	100.9h

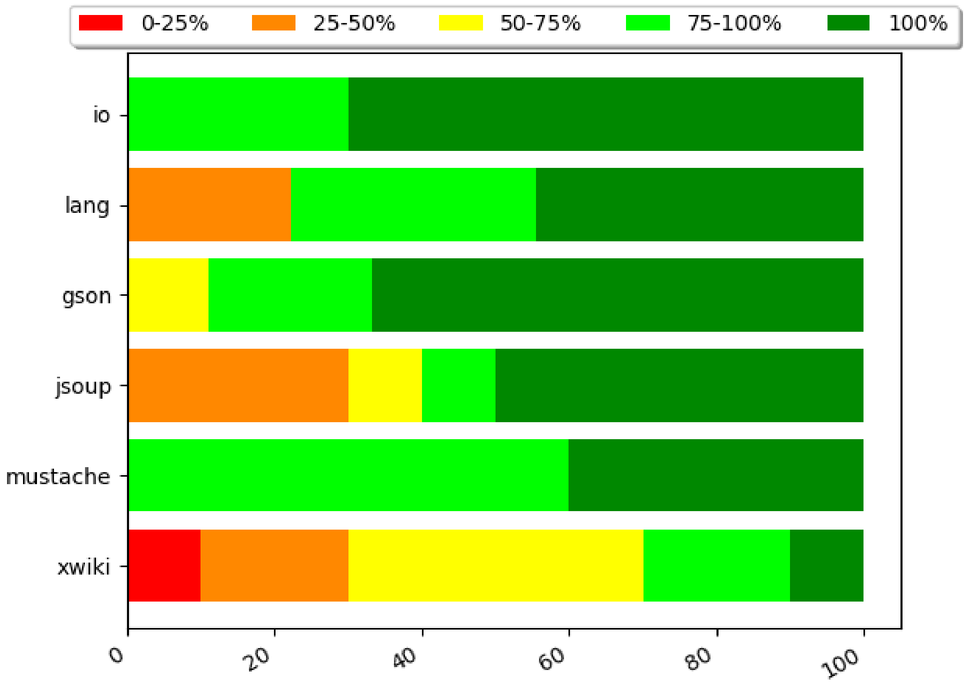
respectively the number of tests modified or added by the commit, and the size of the diff in terms of line additions (in green) and deletions (in red); the seventh and eighth columns are respectively the diff coverage and the number of tests DCI selected; the ninth column provides the amplification results for DCI-*A-Amplification* , and it is either a ✓ with the number of amplified tests that detect a behavioral change or a - if DCI did not succeed in generating a test that detects a change; the tenth column displays the time spent on the amplification phase; The eleventh and the twelfth are respectively a ✓ with the number of amplified tests for DCI-*I-Amplification* (or - if a change is not detected) for 3 iterations. The last row reports the total over the 6 projects. For the tenth and the twelfth columns of the last row, the first number is the number of successes, *i.e.* the number of times DCI produced at least one amplified test method that detects the behavioral change, for DCI-*A-Amplification* and DCI-*I-Amplification* respectively. The numbers between brackets correspond to the total number of amplified test methods that DCI produces in each mode.

5.3.4.1 Characteristics of commits with behavioral changes in the context of continuous integration

This section describes the characteristics of commits introducing behavioral changes in the context of continuous integration. The number of test methods at the time of the commit shows two aspects of our benchmark: 1) there are only strongly tested projects; 2) the number of tests evolve over time due to test evolution. Every commit in the benchmark comes with test modifications (new tests or updated tests), and commit sizes are quite diverse. The three smallest commits are COMMONS-IO#703228A, GSON#44CAD04 and JSOUP#E5210D1 with 6 modifications, and the largest is GSON#45511FD with 334 modifications. Finally, on average, commits have 66.11% coverage. The distribution of diff coverage is reported graphically by Figure 5.3: in commons-io all selected commits have more than 75% coverage. In XWiki-Commons, only 50% of commits have more than 75% coverage. Overall, 31 / 60 commits have at least 75% of the changed lines covered. This validates the correct implementation of the selection criteria that ensures the presence of a test specifying the behavioral change.

Thanks to the selection criteria, a curated benchmark of 50 commits with a behavioral change is available for the evaluation. This benchmark comes from notable open-source projects, and covers a diversity of commit sizes. The benchmark is publicly available and documented for future research on this topic.

Figure 5.3: Distribution of diff coverage per project of our benchmark.



5.3.4.2 RQ1: To what extent are DCI-*A-Amplification* and DCI-*I-Amplification* able to produce amplified test methods that detect the behavioral changes?

The last 4 columns of Table 5.2 are dedicated to **RQ**$_1$. For example, DCI-*A-Amplification* and DCI-*I-Amplification* generated respectively 1 and 39 amplified test methods that detect the behavioral change for COMMONS-IO#F00D97A (4^{th} row). In the other hands, only DCI-*I-Amplification* has been able to obtain amplified test methods for COMMONS-IO#81210EB (8^{th} row).

Overall, DCI-*A-Amplification* generates amplified tests that detect 9 out of 60 behavioral changes. Meanwhile, DCI-*I-Amplification* generates amplified tests that detect 25 out of 60 behavioral changes.

Regarding the number of generated tests. DCI-*I-Amplification* generates a large number of test methods, compared to DCI-*A-Amplification* only (15 versus 6708, see column "total" at the bottom of the table). Both DCI-*A-Amplification* and DCI-*I-Amplification* can generate amplified tests, however since DCI-*A-Amplification* does not produce a large amount of test methods the developers do not have to triage a large set of test cases. Also, since DCI-*A-Amplification* only adds assertions, the amplified tests are easier to understand than the ones generated by DCI-*I-Amplification* .

DCI-*I-Amplification* takes more time than DCI-*A-Amplification* (for successful cases

38.7 seconds versus 3.3 hours on average). The difference comes from the time consumed during the exploration of the input space in the case of DCI-*I-Amplification* , while DCI-*A-Amplification* focuses on the amplification of assertions only, which represents a much smaller space of solutions.

Overall, DCI successfully generates amplified tests that detect a behavioral change in 46% of the commits in our benchmark(25 out of 60). Recall that the 60 commits analyzed are real changes in complex code bases. They represent modifications, sometimes deep in the code, that are challenges with respect to testability [Voas 1995]. Consequently, the fact DCI generates test cases that detect behavioral changes, is considered an achievement. The commits for which DCI fails to detect the change can be considered as a target for future research on this topic.

A successful detection by an amplified test method is analyzed. Commit 3FADFDD[5] from commons-lang has been selected because it is succinct enough to be discussed. The diff is shown in Listing 5.1.

Listing 5.1: Diff of commit 3FADFDD from commons-lang.

```
@@ -2619,7 +2619,7 @@ protected void appendFieldStart(final StringBuffer buffer,
    final String fieldNam

-           super.appendFieldStart(buffer, FIELD_NAME_QUOTE + fieldName
+           super.appendFieldStart(buffer, FIELD_NAME_QUOTE +
+               StringEscapeUtils.escapeJson(fieldName) + FIELD_NAME_QUOTE);
 }
```

The developer added a method call to a method that escapes specials characters in a string. The changes come with a new test method that specifies the new behavior.

DCI starts the amplification from the `testNestingPerson` test method defined in `JsonToStringStyleTest` showed in Listing 5.2.

The test is selected for amplification because it triggers the execution of the changed line.

The resulting amplified test method is shown in Listing 5.3. From this test method, DCI generates an amplified test method shown in Listing 5.3. In this generated test, SBAMPL applies 2 input transformations: 1 duplication of method call and 1 character replacement in an existing String literal. The latter transformation is the key transformation: DCI replaced an 's' inside "person" by '/' resulting in "per/on" where "/" is a special character that must be escaped (Line 2). Then, DCI generated 11 assertions, based on the modified inputs. The amplified test the behavioral change: in the pre-commit version, the expected value is: `"{`

[5] https://github.com/apache/commons-lang/commit/3fadfdd

Listing 5.2: Selected test method as a seed to be amplified for commit 3FADFDD from commons-lang.

```
@Test
public void testPerson() {
 final Person p = new Person();
 p.name = "Jane Doe";
 p.age = 25;
 p.smoker = true;

 assertEquals(
  "{\"name\":\"Jane Doe\",\"age\":25,\"smoker\":true}",
  new ToStringBuilder(p).append("name", p.name)
  .append("age", p.age).append("smoker", p.smoker)
  .toString()
 );
}
```

Listing 5.3: Test generated by DCI that detects the behavioral change of 3FADFDD from commons-lang.

```
@Test(timeout = 10000)
public void testPerson_literalMutationString85602() throws Exception {
 final ToStringStyleTest.Person p = new ToStringStyleTest.Person();
 p.name = "Jane Doe";
 Assert.assertEquals("Jane Doe", p.name);
 p.age = 25;
 p.smoker = true;
 String o_testPerson_literalMutationString85602__6 =
  new ToStringBuilder(p)
   .append("n/me", p.name)
   .append("age", p.age)
   .append("smoker", p.smoker)
   .toString();
 Assert.assertEquals(
  "{\"n/me\":\"Jane Doe\",\"age\":25,\"smoker\":true}",
  o_testPerson_literalMutationString85602__6
 );
 Assert.assertEquals("Jane Doe", p.name);
}
```

`...  per/on":{"name":"Jane Doe" ...}"` while in the post-commit version it is `"{ ...  per\/on":{"name":"Jane Doe" ...}"` (Line 3).

> **RQ1**: Overall, DCI detects the behavioral changes in a total of 25/60 commits. Individually, DCI-*I-Amplification* finds changes in 25/60, while DCI-*A-Amplification* in 9/60 commits. Since DCI-*I-Amplification* also uses AAMPL to generate assertions, all DCI-*A-Amplification* 's commits are contained in DCI-*I-Amplification* 's. However, the search-based algorithm, through exploration, finds many more behavioral changes, making it more effective albeit at the cost of execution time.

5.3.4.3 RQ2: What is the impact of the number of iteration performed by DCI-*I-Amplification* ?

The results are reported in Table 5.3 This table can be read as follow: the first column is the name of the project; the second column is the commit identifier; then, the third, fourth, fifth, sixth, seventh and eighth provide the amplification results and execution time for each number of iteration 1, 2, and 3. A ✓ indicates with the number of amplified tests that detect a behavioral change and a - denotes that DCI did not succeed in generating a test that detects a change. The last row reports the total over the 6 projects. For the third, fifth and the seventh columns of the last row, the first number is the number of successes, *i.e.* the number of times that DCI produced at least one amplified test method that detect the behavioral change, for respectively$iteration = 1$, $iteration = 2$ and $iteration = 3$. The numbers in parentheses are the total number of amplified test methods that DCI produces with each number of iteration.

Overall, DCI-*I-Amplification* generates amplified tests that detect 23, 24, and 25 out of 60 behavioral changes for respectively $iteration = 1$, $iteration = 2$ and $iteration = 3$. The more iteration DCI-*I-Amplification* does, the more it explores, the more it generates amplified tests that detect the behavioral changes but the more it takes time also. When DCI-*I-Amplification* is used with $iteration = 3$, it generates amplified test methods that detect 2 more behavioral changes than when it is used with $iteration = 1$ and 1 more than when it is used with $iteration = 2$.

In average, DCI-*I-Amplification* generates 18, 53, and 116 amplified tests for respectively $iteration = 1$, $iteration = 2$ and $iteration = 3$. This number increases by 544% from $iteration = 1$ to $iteration = 3$. This increase is explained by the fact that DCI-*I-Amplification* explores more with more iteration and thus is able to generate more amplified test methods that detect the behavioral changes.

In average DCI-*I-Amplification* takes 23, 64, and 105 minutes to perform the amplification for respectively $iteration = 1$, $iteration = 2$ and $iteration = 3$. This number increases by 356% from $iteration = 1$ to $iteration = 3$.

Table 5.3: Evaluation of the impact of the number of iteration done by DCI-*I-Amplification* on 60 commits from 6 open-source projects.

	id	$it = 1$	Time	$it = 2$	Time	$it = 3$	Time
commons-io	c6b8a38	0	25.0s	0	62.0s	0	98.0s
	2736b6f	✓ (1)	26.1m	✓ (2)	44.2m	✓ (12)	76.3m
	a4705cc	0	4.1m	0	21.1m	0	38.1m
	f00d97a	✓ (7)	13.0s	✓ (28)	19.0s	✓ (39)	27.0s
	3378280	✓ (6)	15.0s	✓ (10)	20.0s	✓ (11)	24.0s
	703228a	0	30.3m	0	55.1m	0	71.0m
	a7bd568	0	28.6m	0	52.0m	0	65.2m
	81210eb	✓ (2)	14.0s	✓ (4)	18.0s	✓ (8)	23.0s
	57f493a	0	20.0s	0	32.0s	0	54.0s
	5d072ef	✓ (461)	32.2m	✓ (1014)	65.5m	✓ (1538)	2.2h
	total	477	2.0h	1058	4.0h	1608	6.5h
	average	47.70	12.3m	105.80	24.0m	160.80	38.8m
commons-lang	f56931c	0	0.0s	0	3.7m	0	8.5m
	87937b2	0	3.5m	0	10.5m	0	18.1m
	09ef69c	0	97.0s	0	21.0m	0	98.8m
	3fadfdd	✓ (1)	2.0m	✓ (1)	9.3m	✓ (4)	17.2m
	e7d16c2	✓ (3)	111.0s	✓ (2)	8.4m	✓ (2)	15.1m
	50ce8c4	✓ (61)	38.0s	✓ (97)	78.0s	✓ (135)	2.0m
	2e9f3a8	0	11.4m	0	35.0m	0	66.5m
	c8e61af	0	16.0s	0	16.0s	0	16.0s
	d8ec011	0	36.0s	0	68.0s	0	2.3m
	7d061e3	0	79.0s	0	5.8m	0	11.4m
	total	65	23.3m	100	96.4m	141	4.0h
	average	6.50	2.3m	10.00	9.6m	14.10	24.0m
gson	b1fb9ca	0	14.6m	0	51.0m	0	92.5m
	7a9fd59	✓ (7)	33.0s	✓ (48)	73.0s	✓ (108)	2.1m
	03a72e7	0	30.2m	0	102.3m	0	3.2h
	74e3711	0	6.0s	0	11.0s	0	16.0s
	ada597e	0	61.0s	0	4.9m	0	8.7m
	a300148	0	45.2m	✓ (4)	2.6h	✓ (6)	4.9h
	9a24219	0	10.8m	0	28.4m	0	48.9m
	9e6f2ba	0	79.0s	0	4.5m	✓ (2)	8.5m
	44cad04	✓ (4)	21.0s	✓ (21)	30.0s	✓ (37)	40.0s
	b2c00a3	0	31.5m	0	111.8m	0	3.6h
	total	11	2.3h	73	7.7h	153	14.4h
	average	1.10	13.6m	7.30	46.0m	15.30	86.5m
jsoup	426ffe7	✓ (126)	5.4m	✓ (172)	19.2m	✓ (198)	33.6m
	a810d2e	0	90.0s	0	13.9m	0	26.6m
	6be19a6	0	8.1m	0	39.7m	0	67.7m
	e38dfd4	0	117.0s	0	6.3m	0	12.5m
	e9feec9	0	20.0s	0	50.0s	0	95.0s
	0f7e0cc	✓ (1)	2.4h	✓ (7)	6.8h	✓ (36)	11.8h
	2c4e79b	0	7.1m	0	34.1m	0	4.7h
	e5210d1	0	45.0s	0	2.3m	0	4.9m
	df272b7	0	43.0s	0	2.2m	0	4.6m
	3676b13	✓ (6)	21.4m	✓ (35)	2.9h	✓ (52)	6.8h
	total	133	3.2h	214	11.6h	286	25.8h
	average	13.30	19.4m	21.40	69.8m	28.60	2.6h
mustache.java	a1197f7	✓ (28)	5.9h	✓ (124)	8.4h	✓ (204)	10.1h
	8877027	0	30.5m	0	58.4m	0	100.2m
	d8936b4	0	3.2m	0	4.8m	0	84.2m
	88718bc	✓ (13)	78.0s	✓ (85)	2.5m	✓ (149)	3.7m
	339161f	✓ (143)	115.9m	✓ (699)	4.1h	✓ (1312)	5.8h
	774ae7a	✓ (18)	2.7m	✓ (65)	4.7m	✓ (124)	6.8m
	94847cc	✓ (122)	5.3h	✓ (580)	10.4h	✓ (2509)	21.4h
	eca08ca	0	8.1m	0	24.3m	0	41.8m
	6d7225c	0	7.9m	0	26.8m	0	40.1m
	8ac71b7	✓ (2)	2.7m	✓ (48)	3.8m	✓ (124)	5.6m
	total	326	14.0h	1601	25.0h	4422	42.0h
	average	32.60	84.3m	160.10	2.5h	442.20	4.2h
xwiki-commons	ffc3997	0	19.0s	0	18.0s	0	18.0s
	ced2635	0	8.0m	0	31.8m	0	2.5h
	10841b1	0	56.2m	0	2.9h	0	3.4h
	848c984	0	18.0s	0	17.0s	0	18.0s
	adfefec	✓ (22)	3.5m	✓ (57)	9.9m	✓ (3)	14.9m
	d3101ae	✓ (9)	11.6m	✓ (12)	28.2m	✓ (31)	41.4m
	a0e8b77	✓ (10)	12.0m	✓ (17)	28.2m	✓ (60)	42.1m
	78ff099	✓ (4)	2.6m	✓ (4)	4.6m	✓ (4)	6.6m
	1b79714	0	4.0m	0	10.7m	0	17.9m
	6dc9059	0	4.0m	0	10.8m	0	20.5m
	total	45	102.8m	90	4.9h	98	8.2h
	average	4.50	10.3m	9.00	29.7m	9.80	49.5m
	total	22(1057)	23.7h	23(3136)	54.9h	24(6708)	100.9h

Table 5.4: Number of successes, *i.e.* DCI produced at least one amplified test method that detects the behavioral changes, for 11 different seeds.

Seed	ref	1	2	3	4	5	6	7	8	9
#Success	23	18	17	17	17	19	21	18	21	18

Impact of the randomness The number of amplified test methods obtained by the different seeds are reported in Table 5.4.

This table can be read as follow: the first column is the id of the commit. the second column is the result obtained with the default seed, used during the evaluation for $\mathbf{RQ}_1$. the ten following columns are the results obtained for the 10 different seeds.

The computed confidence interval is $[20.34, 17.66]$ It means that, from our samples, with probability 0.95, the real value of the number of successes lies in this interval.

Answer to **RQ2**: DCI-*I-Amplification* detects 23, 24, and 25 behavioral changes out of 60 for respectively $iteration = 1$, $iteration = 2$ and $iteration = 3$. The number of iteration done by DCI-*I-Amplification* impacts the number of behavioral changes detected, the number of amplified test methods obtained and the execution time.

5.3.4.4 RQ3: What is the effectiveness of our test selection method?

To answer **RQ3**, there is no quantitative approach to take, because there is no ground-truth data or metrics to optimize. Per the protocol described in subsection 5.3.3, the answer to this question is based on manual analysis: 1 commit per project is randomly selected. Then the relevance of the selected tests for amplification is analyzed.

Following an example, in order to give an intuition of what are the characteristics of the test selection for amplification to be relevant. The selection is considered relevant If `TestX` is selected for amplification, following a change to method `X`. The key is that DCI will generate an amplified test `TestX'` that is a variant of `TestX`, and, consequently, the developer will directly get the intention of the new test `TestX'` and what behavioral change it detects.

COMMONS-IO#C6B8A38[6]: the test selection returns 3 test methods: `testContentEquals`, `testCopyURLToFileWithTimeout` and `testCopyURLToFile` from the same test class: `FileUtilsTestCase`. The considered commit modifies the method `copyToFile` from `FileUtils`. There is a link between the changed file and the intention of 2 out 3 tests to be amplified. The selection is thus considered relevant.

COMMONS-LANG#F56931C[7]: the test selection returns 39 test methods from 5 test classes: `FastDateFormat_ParserTest`, `FastDateParserTest`, `DateUtil-`

[6] https://github.com/apache/commons-io/commit/c6b8a38
[7] https://github.com/apache/commons-lang/commit/f56931c

sTest, FastDateParser_TimeZoneStrategyTest and FastDateParser_-MoreOrLessTest. This commit modifies the behavior of two methods: simple-Quote and setCalendar of class FastDateParser. When manually analyzed, it reveals two intentions: 1) test behaviors related to parsing; 1) test behaviors related to dates. While this is meaningful, a set of 39 methods is clearly not a focused selection, not as focused as for the previous example. The selection can be considered relevant, but not focused.

GSON#9E6F2BA[8]: the test selection returns 9 test methods from 5 different test classes. 3 out of those 5 classes JsonElementReaderTest, JsonReaderPathTest and JsonParserTest relate to the class modified in the commit(JsonTreeReader). The selection is thus considered relevant but unfocused.

JSOUP#E9FEEC9[9], the test selection returns the 4 test methods defined in the Xml-TreeBuilderTest class caseSensitiveDeclaration, handlesXmlDecla-rationAsDeclaration, testDetectCharsetEncodingDeclaration and testParseDeclarationAttributes. The commit modifies the behavior of the class XmlTreeBuilder. Here, the test selection is relevant. Actually, the ground-truth manual test added in the commit is also in the XmlTreeBuilderTest class. If DCI proposes a new test there to capture the behavioral change, the developer will understand its relevance and its relation to the change.

MUSTACHE.JAVA#88718BC[10], the test selection returns the testInvalidDelimiters test method defined in the com.github.mustachejava.InterpreterTest test class. The commit improves an error message when an invalid delimiter is used. Here, the test selection is relevant since it selected testInvalidDelimiters which is the dedicated test to the usage of the test invalid delimiters. This ground-truth test method is also in the test class com.github.mustachejava.InterpreterTest.

XWIKI-COMMONS#848C984[11] the test selection returns a single test method cre-ateReference from test class XWikiDocumentTest. The main modification of this commit is on class XWikiDocument. Since XWikiDocumentTest is the test class dedicated to XWikiDocument, the selection is considered relevant.

[8] https://github.com/google/gson/commit/9e6f2ba
[9] https://github.com/jhy/jsoup/commit/e9feec9
[10] https://github.com/spullara/mustache.java/commit/88718bc
[11] https://github.com/xwiki/xwiki-commons/commit/848c984

Listing 5.4: Test generated by DCI-*I-Amplification* that detects the behavioral change introduced by commit 81210EB in commons-io.

```
@Test(timeout = 10000)
public void readMulti_literalMutationNumber3() {
 BoundedReader mr = new BoundedReader(sr, 0);
 char[] cbuf = new char[4];
 for (int i = 0; i < (cbuf.length); i++) {
  cbuf[i] = 'X';
 }
 final int read = mr.read(cbuf, 0, 4);
 Assert.assertEquals(0, ((int) (read)));
}
```

> Answer to **RQ3**: In 4 out of 6 of the manually analyzed cases, the tests selected to be amplified relate, semantically, to the modified application code. In the 2 remaining cases, it selected over and above the tests to be amplified. That is, it selects tests whose intention is semantically pertinent to the change, but it also includes tests that are not. However, even in this case, DCI's test selection provides developers with important and targeted context to better understand the behavioral change at hand.

5.3.4.5 RQ4: How do human and generated tests that detect behavioral changes differ?

When DCI generates an amplified test method that detects the behavioral change, it can be compared to the ground truth version (the test added in the commit) to see whether it captures the same behavioral change. For each project, I select 1 successful application of DCI, and compare the DCI test against the human test. If they capture the same behavioral change, it means they have the same intention and the amplification is considered as a success.

COMMONS-IO#81210EB[12]: This commit modifies the behavior of the `read()` method in `BoundedReader`. Listing 5.4 shows the test generated by DCI-*I-Amplification* . This test is amplified from the existing `readMulti` test, which indicates that the intention is to test the read functionality. The first line of the test is the construction of a `BoundedReader` object which is also the class modified by the commit. DCI-*I-Amplification* modified the second parameter of the constructor call (transformed 3 into a 0) and generated two assertions (only 1 is shown). The first assertion, associated to the new test input, captures the behavioral difference. Overall, this can be considered as a successful amplification.

[12] https://github.com/apache/commons-io/commit/81210eb

Listing 5.5: Developer test for commit 81210EB of commons-io.

```
@Test(timeout = 5000)
public void testReadBytesEOF() {
 BoundedReader mr = new BoundedReader( sr, 3 );
 BufferedReader br = new BufferedReader( mr );
 br.readLine();
 br.readLine();
}
```

Listing 5.6: Test generated by DCI-*I-Amplification* that detects the behavioral change of E7D16C2 in commons-lang.

```
@Test(timeout = 10000)
public void testAppendSuper_literalMutationString64() {
  String o_testAppendSuper_literalMutationString64__15 =
   new ToStringBuilder(base)
    .appendSuper((((("Integer@8888[" + (System.lineSeparator()))
       + "  null")
     + (System.lineSeparator())) + "]"))
    .append("a", "b0/|]")
    .toString();
 Assert.assertEquals("{\"a\":\"b0/|]\"}",
     o_testAppendSuper_literalMutationString64__15);
}
```

Now, look at the human test contained in the commit, shown in Listing 5.5. It captures the behavioral change with the timeout (the test timeouts on the pre-commit version and goes fast enough on the post-commit version). Furthermore, it only indirectly calls the changed method through a call to `readLine`.

In this case, the DCI test can be considered better than the developer test because 1) it relies on assertions and not on timeouts, and 2) it directly calls the changed method (`read`) instead of indirectly.

COMMONS-LANG#E7D16C2[13]: this commit escapes special characters before adding them to a `StringBuffer`. Listing 5.6 shows the amplified test method obtained by DCI-*I-Amplification* . The assertion at the bottom of the excerpt is the one that detects the behavioral change. This assertion compares the content of the `StringBuilder` against an expected string. In the pre-commit version, no special character is escaped, *e.g.*'\n'. In the post-commit version, the amplified test fails since the code now escapes the special character \.

[13] https://github.com/apache/commons-lang/commit/e7d16c2

Listing 5.7: Developer test for E7D16C2 of commons-lang.

```
@Test
public void testLANG1395() {
 assertEquals("{\"name\":\"value\"}",
  new ToStringBuilder(base).append("name","value").toString());
 assertEquals("{\"name\":\"\"}",
  new ToStringBuilder(base).append("name","").toString());
 assertEquals("{\"name\":\"\\\"\"}",
  new ToStringBuilder(base).append("name",'"').toString());
 assertEquals("{\"name\":\"\\\\\"}",
  new ToStringBuilder(base).append("name",'\\').toString());
 assertEquals("{\"name\":\"Let's \\\"quote\\\" this\"}",
  new ToStringBuilder(base).append("name","Let's \"quote\" this")
      .toString());
}
```

Let's have a look to the human test method shown in Listing 5.7. Here, the developer specified the new escaping mechanism with 5 different inputs.

The main difference between the human test and the amplified test is that the human test is more readable and uses 5 different inputs. However, the amplified test generated by DCI is valid since it detects the behavioral change correctly.

GSON#44CAD04[14]: This commit allows Gson to deserialize a number represented as a string. Listing 5.8 shows the relevant part of the test generated by DCI-*I-Amplification* , based on `testNumberDeserialization` of `PrimitiveTest` as a seed. The DCI test detects the behavioral changes at lines 3 and 4. On the pre-commit version, line 3 throws a `JsonSyntaxException`. On the post-commit version, line 4 throws a `NumberFormatException`. In other words, the behavioral change is detected by a different exception (different type and not thrown at the same line). [15].

The amplified test methods is now compared against the developer-written ground-truth method, shown in Listing 5.9. This short test verifies that the program handles a number-as-string correctly. For this example, the DCI test does indeed detect the behavioral change, but in an indirect way. On the contrary, the developer test is shorter and directly targets the changed behavior, which is better.

JSOUP#3676B13[16]: This change is a pull request (*i.e.* a set of commits) and introduces 5 new behavioral changes. There are two improvements: skip the first new lines in pre tags

[14] https://github.com/google/gson/commit/44cad04

[15] Interestingly, the number is parsed lazily, only when needed. Consequently, the exception is thrown when invoking the `longValue()` method and not when invoking `parse()`

[16] https://github.com/jhy/jsoup/commit/3676b13

Listing 5.8: Test generated by DCI that detects the behavioral change of commit 44CAD04 in Gson.

```
public void
    testNumberDeserialization_literalMutationString8_failAssert0()
     throws Exception {
 try {
  String json = "dhs";
  actual = gson.fromJson(json, Number.class);
  actual.longValue();
  junit.framework.TestCase.fail(
   "testNumberDeserialization_literalMutationString8 should have
      thrown JsonSyntaxException");
 } catch (JsonSyntaxException expected) {
  TestCase.assertEquals("Expecting number, got: STRING", expected
     .getMessage());
 }
}
```

Listing 5.9: Provided test by the developer for 44CAD04 of Gson.

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

Listing 5.10: Test generated by DCI-*I-Amplification* that detects the behavioral change of 3676B13 of Jsoup.

```
@Test(timeout = 10000)
public void parsesBooleanAttributes_add4942() {
 String html = "<a normal=\"123\" boolean empty=\"\"></a>";
 Element el = Jsoup.parse(html).select("a").first();
 List<Attribute> attributes = el.attributes().asList();
 Attribute o_parsesBooleanAttributes_add4942__15 =
 attributes.get(1);
 Assert.assertEquals("boolean=\"\"",
   ((BooleanAttribute) (o_parsesBooleanAttributes_add4942__15)).
      toString());
}
```

Listing 5.11: Provided test by the developer for 3676B13 of Jsoup.

```
@Test
public void booleanAttributeOutput() {
 Document doc = Jsoup.parse("<img src=foo noshade='' nohref async
     =async autofocus=false>");
 Element img = doc.selectFirst("img");

 assertEquals("<img src=\"foo\" noshade nohref async autofocus=\"
     false\">", img.outerHtml());
}
```

and support deflate encoding, and three bug fixes: throw exception when parsing some URLs, add spacing when output text, and no collapsing of attribute with empty values. Listing 5.10 shows an amplified test obtained using DCI-*I-Amplification* . This amplified test has 15 assertions and a duplication of method call. Thanks to this duplication and these generated assertions on the `toString()` method, this test is able to capture the behavioral change introduced by the commit.

As before, the amplified test method is compared to the developer's test. The developer uses the `Element` and `outerHtml()` methods rather than `Attribute` and `toString()`. However, the method `outerHtml()` in `Element` will call the `toString()` method of `Attribute`. For this behavioral change, it concerns the `Attribute` and not the `Element`. So, the amplified test is arguably better, since it is closer to the change than the developer's test. But, DCI-*I-Amplification* generates amplified tests that detect 2 of 5 behavioral changes: adding spacing when output text and no collapsing of attribute with empty values only, so regarding the quantity of changes, the human tests are more complete.

Listing 5.12: Test generated by DCI-*I-Amplification* that detects the behavioral change of 774AE7A of Mustache.java.

```
@Test(timeout = 10000)
public void
    getReaderNullRootDoesNotFindFileWithAbsolutePath_litStr4() {
 ClasspathResolver underTest = new ClasspathResolver();
 Reader reader = underTest.getReader(" does not exist");
 Assert.assertNull(reader);
 Matcher<Object>
 o_getReaderNullRootDoesNotFindFileWithAbsolutePath_litStr4__5 =
  Is.is(CoreMatchers.nullValue());
 Assert.assertEquals("is null",
  ((Is) (
     o_getReaderNullRootDoesNotFindFileWithAbsolutePath_litStr4__5
     ))
  .toString()
 );
 Assert.assertNull(reader);
}
```

MUSTACHE.JAVA#774AE7A[17]: This commit fixes an issue with the usage of a dot in a relative path on Window in the method `getReader` of class `ClasspathResolver`. The test method `getReaderNullRootDoesNotFindFileWithAbsolutePath` has been used as seed by DCI. It modifies the existing string literal with another string used somewhere else in the test class and generates 3 new assertions. The behavioral change is detected thanks to the modified strings: it produces the right test case containing a space.

The developer proposed two tests that verify that the object reader is not null when getting it with dots in the path. There are shown in Listing 5.13. These tests invoke the method `getReader` which is the modified method in the commit. The difference is that the DCI-*I-Amplification* 's amplified test method provides a non longer valid input for the method `getReader`. However, providing such inputs produce errors afterward which signal the behavioral change. In this case, the amplified test is complementary to the human test since it verifies that the wrong inputs are no longer supported and that the system immediately throws an error.

XWIKI-COMMONS#D3101AE[18]: This commit fixes a bug in the `merge` method of class `DefaultDiffManager`. Listing 5.14 shows the amplified test method obtained by DCI-*A-Amplification* . DCI used `testMergeCharList` as a seed for the amplification process, and generates 549 new assertions. Among them, 1 assertion captures

[17] https://github.com/spullara/mustache.java/commit/774ae7a
[18] https://github.com/xwiki/xwiki-commons/commit/d3101ae

Listing 5.13: Developer test for 774AE7A of Mustache.java.

```
@Test
public void getReaderWithRootAndResourceHasDoubleDotRelativePath() throws Exception {
 ClasspathResolver underTest = new ClasspathResolver("templates");
 Reader reader = underTest.getReader("absolute/../absolute_partials_template.html");
 assertThat(reader, is(notNullValue()));
}

@Test
public void getReaderWithRootAndResourceHasDotRelativePath() throws Exception {
 ClasspathResolver underTest = new ClasspathResolver("templates");
 Reader reader = underTest.getReader("absolute/./nested_partials_sub.html");
 assertThat(reader, is(notNullValue()));
}
```

the behavioral change between the two versions of the program: "assertEquals(0, result.getLog().getLogs(LogLevel.ERROR).size());". The behavioral change that is detected is the presence of a new logging statement in the diff. After verification, there is indeed such a behavioral change in the diff, with the addition of a call to "logConflict" in the newly handled case.

The developer's test is shown in Listing 5.15. This test method directly calls method `merge`, which is the method that has been changed. What is striking in this test is the level of clarity: the variable names, the explanatory comments and even the vertical space formatting are impossible to achieve with DCI-*A-Amplification* and makes the human test clearly of better quality but also longer to write.

Yet, DCI-*A-Amplification* 's amplified tests capture a behavioral change that was not specified in the human test. In this case, amplified tests can be complementary.

> Answer to **RQ4**: In 3 of 6 cases, the DCI test is complementary to the human test. In 1 case, the DCI test can be considered better than the human test. In 2 cases, the human test is better than the DCI test. Even though human tests can be better, DCI can be complementary and catch missed cases, or can provide added-value when developers do not have the time to add a test.

Listing 5.14: Test generated by DCI-*A-Amplification* that detects the behavioral change of D3101AE of XWiki.

```
@Test(timeout = 10000)
public void testMergeCharList() throws Exception {
 MergeResult<Character> result;
 result = this.mocker.getComponentUnderTest()
  .merge(AmplDefaultDiffManagerTest.toCharacters("a"),
   AmplDefaultDiffManagerTest.toCharacters(""),
   AmplDefaultDiffManagerTest.toCharacters("b"),
   null
  );
 int o_testMergeCharList__9 = result.getLog().getLogs(LogLevel.
     ERROR).size();
 Assert.assertEquals(1, ((int) (o_testMergeCharList__9)));
 List<Character> o_testMergeCharList__12 =
     AmplDefaultDiffManagerTest.toCharacters("b");
 Assert.assertTrue(o_testMergeCharList__12.contains('b'));
 result.getMerged();
 result = this.mocker.getComponentUnderTest()
  .merge(AmplDefaultDiffManagerTest.toCharacters("bc"),
   AmplDefaultDiffManagerTest.toCharacters("abc"),
   AmplDefaultDiffManagerTest.toCharacters("bc"),
   null
  );
 int o_testMergeCharList__21 = result.getLog().getLogs(LogLevel.
     ERROR).size();
 Assert.assertEquals(0, ((int) (o_testMergeCharList__21)));
}
```

Listing 5.15: Developer test for D3101AE of XWiki.

```java
@Test
public void testMergeWhenUserHasChangedAllContent() throws Exception
{
 MergeResult<String> result;

 // Test 1: All content has changed between previous and current
 result = mocker.getComponentUnderTest().merge(Arrays.asList("Line 1", "Line 2", "Line 3"),
 Arrays.asList("Line 1", "Line 2 modified", "Line 3", "Line 4 Added"),
 Arrays.asList("New content", "That is completely different"), null);

 Assert.assertEquals(Arrays.asList("New content", "That is completely different"), result.getMerged());

 // Test 2: All content has been deleted
 // between previous and current
 result = mocker.getComponentUnderTest().merge(Arrays.asList("Line 1", "Line 2", "Line 3"),
 Arrays.asList("Line 1", "Line 2 modified", "Line 3", "Line 4 Added"),
 Collections.emptyList(), null);

 Assert.assertEquals(Collections.emptyList(), result.getMerged());
}
```

5.4 Discussion about the scope of DCI

In this section, we overview the current scope of DCI and the key challenges that limit DCI.

Focused applicability From the benchmark, DCI is applicable to limited proportion of commits: 9.93% of the commits analyzed on average. This low proportion is the first limit of DCI usage. However, Once DCI is setup, there is no manual overhead. Even if DCI is not used at each commit, it costs nothing more.

Adoption The evaluation showed that DCI is able to obtain amplified test methods that detect a behavioral change. But, it does not provide any evidence on the fact that developers would exploit such test method. However, from the previous chapter Chapter 4, software developers value the amplified test methods. This provides strong evidence on the potential adoption of DCI.

Performance From our experiments, we see that the time to complete the amplification is the main limitation of DCI. For example DCI took almost 5 hours on JSOUP#2C4E79B, with no result. For the sake of our experimentation, we choose to use a pre-defined number of iteration to bound the exploration. In practice, we recommend to set a time budget (*e.g.* at most one hour per pull-request).

Importance of test seeds By construction, DCI's effectiveness is correlated to the test methods used as seed. For example, see the row of `commons-lang#c8e61af` in Table 5.3, where one can observe that whatever the number of iteration, DCI takes the same time to complete the amplification. The reason is that the seed tests are only composed of assertions statements. Such tests are bad seeds for DCI, and they prevent any good input amplification. Also, DCI requires to have at least one test method that executes the code changes. If the project is poorly tested and does not have any test method that execute the code changes, DCI cannot be applied.

False positives The risk of false positives is a potential limitation of our approach. A false positive would be an amplified test method that passes or fails on both versions, which means that the amplified test method does not detect the behavioral difference between both versions. I manually analyzed 6 commits and none of them are false positives. This increases our confidence that DCI produces a limited number of such confusing test methods.

5.5 Threats to validity

An internal threat is the potential bugs in the implementation of DCI. However, it is heavily tested, with JUnit test suite to mitigate this threat.

In the benchmark, there are 60 commits. The result may be not be generalizable to all programs. But real and diverse applications from GitHub have been carefully selected, all having a strong test suite. The benchmark reflects real programs, and provides an high confidence in the result.

The experiments are stochastic, and randomness is a threat accordingly. To mitigate this threat, I have computed a confidence interval that estimates the number of successes that DCI would obtain.

5.6 Conclusion

This chapter presented the evaluation of DCI, which aim at setting up DSpot inside the CI. The goal of DCI is to produce automatically test methods that detect a behavioral change, *i.e.* a behavioral difference between two versions of the same program. This is done by selecting test methods that execute the changes, then amplify them with DSpot. In addition to this, DSpot keeps amplified test methods that detect the behavioral change, approximate by the fact that the amplified test methods pass on the pre-changes version but fail on the post-change version of the program.

DCI can be integrated to the continuous integration to achieve two major tasks:

1) DCI can improve the regression detection ability of the test suite with respect to a changes. When the behavioral changes highlighted by the test suite is not desired, it means that the developers introduced a regression or a new bug inside the program. Using these amplified test methods, the developers can identify and fix the problem faster than without it. Thus it prevents the merge of hidden bug that could cost a lot of money if users face it when using the application.

2) DCI can help the developers to make evolve the test suite by providing amplified test methods that detect the behavioral change. When this behavioral change is the one desired, the developers just need to negate, manually or using automatic approach such as ReAssert [Daniel 2009a], the assertions. The developer will obtain amplified test methods that strengthen the test suite according to her change.

To evaluate DCI, I selected 60 commits that introduce behavioral changes from 6 open-source projects, from past-evaluation [Vera-Pérez 2018b, Danglot 2018]. Then, I executed DCI on these 60 commits and observe whether or not DCI can generate amplified test methods that detect the behavioral changes. For these 60 commits, DCI has been able to detect 25 behavioral change. To the best of my knowledge, this is the first benchmark of real behavioral changes from open-source projects. This evaluation showed that DCI is able to generate amplified test methods to detect real behavioral changes, introduced by commits. This means that DCI can be used in an industrial context since the selected changes are complex and required deep knowledge of the application, and the selected application are wide-spread and used programs across the open-source community.

In the next chapter, I expose 3 transverval contributions of this thesis:

First, the study of the correctness of program under runtime perturbation: In the previous chapter Chapter 3 and this chapter, I evaluated DSpot in two different contexts: offline amplification and amplification in the CI. DSpot generates amplified test methods, using the state of the program as oracle to build assertions. It means that the current behavior of the program is considered correct by DSpot. What does correct mean? In the first transversal contribution, I studied the correctness program. In particular, how do programs behave under run-time perturbation?

Second, the study of pseudo-tested methods; Pseudo-tested methods are source methods that when the body is removed, the whole test suite pass, while some test methods are executing this methods. In Chapter 4, I use mutation score to measure test suite's quality. The detection of pseudo-tested methods can be done using mutation score and more particularly specifically designed mutants.

And Third, the study of test generation for automatic repair. Automatic repair aims at fixing automatically bugs. One family of automatic repair is the test suite based. This family uses the test suite as oracle to know whether or not the bug have been fix. One can use test generation techniques to enhances these automatic repair techniques and see it as a test amplification process.

CHAPTER 6

Transversal Contributions

Contents

In this chapter, I expose 3 transversal contributions that I made during my thesis. Thanks to the expertise that I developed during my thesis, I worked with my researchers colleagues on side topics yet linked to this thesis.

- The first transversal contribution studies the correctness of program under runtime perturbations in Section 6.1. This work has been done during my master degree.
- The second transversal contribution studies the pseudo-tested methods Section 6.2.
- The third transversal contribution studies patches' overfitting in test-based repair techniques in Section 6.3.

These 3 transversal contributions are supported by published articles [Danglot 2018, Vera-Pérez 2018b, Yu 2019] in *Empirical Software Engineering*. However, the reader can skip this chapter since it is additional materials and this chapter does not expose core contributions.

6.1 Study of Program Correctness

In this thesis, I study the correctness of programs through test methods. DSpot, that has been presented in Chapter 3, is a test suite amplification tool that generates new test methods using the current behavior of the program as oracle to build assertions. In other words, DSpot considers considers the current values that compose the state of the program as **correct**. But, what does correct mean? In this first transversal contribution, I study the correctness program under run-time perturbations.

6.1.1 Problem Statement

Recalling that in Chapter 1, I mentioned a quote of Dijkstra:

"the smallest possible perturbations - i.e. changes of a single bit - can have the most drastic consequences.".

This first study aims at verifying this statement is true or not. Dijkstra considers software as a system that is in an unstable equilibrium, or to put it more precisely, that the correctness of a program output is unstable with respect to perturbations. However, previous works, *e.g.* [Rinard 2005, Li 2007]) suggest the opposite, *i.e.* suggest that programs can accommodate perturbations.

In the context of my thesis, the correctness of programs relies on test suite. However, the incompleteness of test suites make them poor oracles to verify the truthfulness of the Dijkstra's hypothesis.

This is why I devise ATTRACT, an experimental protocol to study the stability of program correctness under execution perturbation. It consists in perturbing the execution of programs according to a perturbation model and observing whether this has an impact on the correctness of the output. The two observable different outcomes are: the perturbation breaks the computation and results in an incorrect output (unstable under perturbation), or the correctness of the output is stable despite the perturbation. This protocol has two key-attributes:

1) It is based on perfect oracle, *i.e.* it verifies the output of the perturbed program is completly correct, *i.e.* bit-wise equals to the output of the unpertubed, or original, program;

2) It explores exhaustively in time and space the perturbation envelop of software.

The remaining of this section is organized as follow: first, I present ATTRACT protocol in subsection 6.1.2. second, I present the experimentation that has been carried out in subsection 6.1.3; and eventually, subsection 6.1.4 conclude this section.

6.1.2 ATTRACT Protocol

To actually perform perturbations, ATTRACT adds **perturbation points** to the program under study where a **perturbation point** (denoted *pp*) is an expression of a given data type. For instance, if one perturbs integer values, the **perturbation points** will be all integer expressions (literals as well as compound expressions). In Listing 6.1[1], there are 3 potential integer perturbation points in a single statement, indicated with braces.

Listing 6.1: Three integer **perturbation points** in a single statement.

$$\texttt{acc |= }\underbrace{i}_{pp_1}\texttt{ >> }\underbrace{mask}_{pp_2};$$

$$\texttt{acc |= }\underbrace{i >> mask}_{pp_3};$$

ATTRACT statically locates **perturbation points** and automatically adds perturbation code around them using a code transformation. The transformation consists of wrapping all compatible expressions into a function call p (for "perturb")[2].

Listing 6.2: The same statement with perturbation code injected.

```
acc |= p(3, p(1,i) >> p(2,mask));
```

In Listing 6.2, each integer expression of Listing 6.1 is wrapped into a call to function p. Function p takes two parameters: a unique index to identify the perturbation point and the expression being perturbed. If p returns the second parameter, the transformed program is semantically equivalent to the original program. The identifier argument enables ATTRACT to perturb only one location at a time. In this example, this identifier ranges from 1 to 3 corresponding to the index given in Listing 6.1 under perturbation point pp.

6.1.2.1 Core Algorithm

The goal of this algorithm is to systematically explore the perturbation space. rst records the number of executions of each perturbation point for each input in a matrix R_{ref} (for

[1] | is the bitwise or operator. $>>$ is the binary right shift operator. The assignment $|=$ is the bitwise or operator between the left operand and the right operand, then the result is affected to the left operand.

[2] In the experimentation, it is implemented on Java programs using the Spoon transformation library [Pawlak 2015].

reference run) without injecting any perturbation. $R_{ref}[pp, i]$ refers to the number of executions of perturbation point pp for a given input i. Then, it re-executes the program for each input, with one perturbation for each point so that each point is perturbed at least and at most once per input. The oracle asserts the correctness of the perturbed execution (output o) for the given input (i). A perturbed execution can have three outcomes: a success, meaning that the correctness oracle validates the output; a failure meaning that the correctness oracle is violated (also called an oracle-broken execution); a runtime error (an exception in Java) meaning that the perturbation has produced a state for which the computation becomes impossible at some point (*e.g.* a division-by-zero).

This algorithm performs a systematic exploration of the perturbation space for a given program and a set of inputs according to a perturbation model.

Algorithm 2 Core Algorithm of ATTRACT.

Require:
$prog$: program,
$model$: perturbation model,
I: set of inputs for program $prog$,
$oracle$: a perfect oracle for program $prog$
Ensure:
exc: counters of execution per perturbation point,
s: counters of success per perturbation point,
ob: counters of oracle broken per perturbation point

```
instrument(prog)
for each input i in I do
    for perturbation point pp in prog do
        R_ref[, i] ← runWithoutPerturbation(prog, i)
        for j = 0, to R_ref[pp, i] do
            o ← runWithPerturbationAt(prog, model, i, pp, j)
            if exception is thrown then
                exc[pp] ← exc[pp] + 1
            else if oracle.assert(i, o) then
                s[pp] ← s[pp] + 1
            else
                ob[pp] ← ob[pp] + 1
            end if
        end for
    end for
end for
```

In Algorithm 2, $R_{ref}[, i]$ denotes column i of matrix R_{ref}. The statement *runWithoutPerturbation(prog, i)* returns a column vector which is assigned to the i^{th} column of matrix R_{ref}; each element is one perturbation point: it contains the number of times each perturbation point is executed in the program *prog* for each input *i*. On the other hand, the statement *runWithPerturbationAt(prog, model, i, pp, j)* runs the program *prog* while using the perturbation model *model*, the perturbation point *pp* at its j^{th} execution for the given input *i*.

6.1.3 Evaluation

The experimentation with a dataset of 10 programs. This dataset has been created following this methodology: first and foremost, the programs can be specified with a perfect oracle; second, they are written in Java; third, they come from diverse application domains in order to maximize external validity.

6.1.3.1 The PONE Experiment

I now present the PONE experiment. Its goal is to explore correctness attraction according to increments ($+1$) of integer values at runtime.

Point that a single perturbation always breaks the output correctness are qualified as **fragile** because a single perturbation at runtime breaks the whole computation. Other points that can be systematically perturbed without any consequence on the correctness of the final output of the program are qualified as **antifragile** (in opposition to fragile). The remainder are in between; those with a correctness ratio larger or equal than 75% are qualified as **robust**.

In the PONE experiment, integer expressions are perturbed. The PONE perturbation model is a small perturbation on an integer expression: a single increment of an integer value only once during an execution. An equivalently small perturbation model is MONE consisting of decrementing integers. An experimentation as been also performed using MONE, however, the results are not reported in this thesis. For more information, see the dedicated article.[Danglot 2018]

Table 6.1: PONE Results. The correctness ratio may not correspond directly to the number of Antifragile and Robust expressions because it is computed over all executions.

Subject	N_{pp}^{int}	|Search space|	# Fragile exp.	#Robust exp.	# Antifragile exp.	Correctness ratio
quicksort	41	151444	6	10	19	77 %
zip	19	38840	5	2	5	76 %
sudoku	89	98211	12	27	8	68 %
md5	164	237680	102	24	7	29 %
rsa	117	2576	55	8	32	54 %
rc4	115	165140	60	7	12	38 %
canny	450	616161	58	129	133	94 %
lcs	79	231786	10	47	13	89 %
laguerre	72	423454	15	24	15	90 %
linreg	75	543720	43	18	11	47 %
total	1221	2509012	366	296	255	66 %

Table 6.1 gives the results of the systematic exploration of the PONE perturbation

space. For each subject, this table gives: the number of integer perturbation points N_{pp}^{int}; the number of perturbed executions (equal to the size of the PONE perturbation space); the number of **fragile** integer expressions; the number of **robust** integer expressions; the number of **antifragile** integer expressions; the **correctness ratio** (percentage of correct outputs) over all perturbed executions.

To sum up, the main conclusions of the PONE experiment are:

- The considered programs are perturbable according to the PONE perturbation model.
- There are very few fully fragile integer expressions in the considered programs.
- There is a majority of highly perturbable integer expressions which results in a high level of correctness attraction.
- Dijkstra's view that software is fragile is not always true, correctness is rather a stable equilibrium than an unstable one.

6.1.4 Discussion

I have devised a protocol called ATTRACT to study the stability of programs under perturbation. ATTRACT exhaustively explores the perturbation space of a given program for a set of inputs according to a perturbation model. An experimentation have been conducted on 10 subjects using the PONE perturbation model. In total, 2509012 perturbed executions have been performed and studied, which makes it one of the largest perturbation experiments ever made. From this experimentation, the presence of "correctness attraction" has been observed. Over all perturbed execution, 66% of them do not break the correctness of the output.

Studying correctness attraction can have divers applicability. One of them is to identify points that can be randomized and protect the software from external and malicious attacks. Also, if one could engineer techniques to automatically improve correctness attraction, in order to obtain zones that accommodate more perturbations of the runtime state, and those zones could be deemed "bug absorbing zones".

To conclude, I imagine two ways to combine both DSpot and ATTRACT: First, using ATTRACT as a test-criterion to amplify test suites, *i.e.* DSpot would keep amplified test methods that detect more perturbations than the original test suite. Second, in ATTRACT, I used perfect oracles. The problem they have been manually devised, and this could be approximated using the test suite. To strengthen it, one could amplifies its test suite with DSpot to have a better approximation of the perfect oracle and thus study deeper the correctness of its program.

6.2 Study of Pseudo-tested Methods

Pseudo-tested methods are source methods that when the body is removed, the whole test suite passes, while some test methods are executing this methods. In Chapter 4, I use mutation score to measure the quality of a test suite. The detection of pseudo-tested methods can be done using mutation score and specifically designed mutants. In this second transversal contributions, we study the presence of pseudo-tested methods and the developers assessment if it is worthy or not to fix them. In DSpot, the default test-criterion used is the pseudo-tested methods, *i.e.* DSpot keeps amplified test methods that specify pseudo-tested methods.

6.2.1 Problem Statement

Niedermayr and colleagues [Niedermayr 2016] recently introduced the concept of *pseudo-tested methods*. These methods are covered by the test suite, but no test case fails even if all behaviors of the method are removed at once, *i.e.* when the body is completely stripped off. This work is novel and intriguing: such pseudo-tested methods are present in all projects, even those with test suites that have high coverage ratio.

If those results hold, it calls for more research on this topic. This is the motivation of this paper: first, we challenge the external validity of Niedermayr *et al.*'s experiment with new study subjects, second we perform an in-depth qualitative empirical study of pseudo-tested methods. In particular, we want to determine if pseudo-tested methods are indicators of badly tested code. While this seems to be intuitively true, we aim at quantifying this phenomenon. Second, we want to know whether pseudo-tested methods are relevant indicators for developers who wish to improve their test suite. In fact, these methods may encapsulate behaviors that are poorly specified by the test suite, but are not relevant functionalities for the project.

6.2.2 Definition and Implementation

Let P be a program and m be a method; $S = \cup_{m \in P} \mathit{effects}(m)$ the set of effects of all methods in P; $\mathit{effects}(m)$ a function $\mathit{effects} : P \rightarrow S$ that returns all the effects of a method m; detect, a predicate $TS \times S \rightarrow \{\top, \bot\}$ that determines if an effect is detected by TS. Here, we consider the following possible effects that a method can produce: change the state of the object on which it is called, change the state of other objects (by calling other methods), return a value as a result of its computation.

Definition 1 *A method is said to be pseudo-tested with respect to a test suite, if the test suite covers the method and does not assess any of its effects.*

A "pseudo-tested" method, as defined previously, is an idealized concept. We now describe an algorithm that implements a practical way of collecting a set of pseudo-tested methods in a program P, in the context of the test suite TS, based on the original proposal of Niedermayr et al. [Niedermayr 2016]. It relies on the idea of "extreme code transformations", which consists in completely stripping out the body of a method.

Algorithm Algorithm 3 starts by analyzing all methods of P that are covered by the test suite and fulfill a predefined selection criterion (predicate ofInterest in line 1). This critetion is based on the structure of the method and aims at reducing the number of false positives detected by the procedure. It eliminates uninteresing methods such as trivial setter and getters or empty void methods. If the method returns a value, the body of the method is stripped out and we generate a few variants that simply return predefined values (line 3).[3] If the method is void, we strip the body without further action (line 7). Once we have a set of variants, we run the test suite on each of them, if no test case fails on any of the variants of a given method, we consider the method as pseudo-tested (line 15). One can notice in line 13 that all extreme transformations are applied to the original program and are analyzed separately.

To conduct our study, we have implemented Algorithm 3 in an open source tool called Descartes[4]. The tool can detect pseudo-tested methods in Java programs tested with a JUnit test suite. Descartes is developed as an extension of Pitest [Coles 2016], and "extreme transformation" can be seen as extreme mutations, in Pitest parlance. It leverages the maturity of Pitest and handles the discovery of points where extreme transformations can be applied and the creation of the new program variants [Vera-Pérez 2018a]. Being open-source, we hope that Descartes will be used by future research on the topic of pseudo-tested methods .

6.2.3 Evaluation

We selected 21 open source projects in a systematic manner to conduct our experiments. We considered active projects written in Java, that use maven as main build system, JUnit as the main testing framework and their code is available in a version control hosting service, mostly GitHub.

6.2.3.1 Frequency of pseudo-testedMethods

With this evaluation, we aim at characterizing the prevalence of pseudo-tested methods. It is a conceptual replication of the work by Niedermayr et al. [Niedermayr 2016], with a

[3] Compared to Niedermayr et al. [Niedermayr 2016], we add two new transformations, one to return $null$ and another to return an empty array. These additions allow to expand the scope of methods to be analyzed.

[4] https://github.com/STAMP-project/pitest-descartes

Algorithm 3 Procedure to detect pseudo-tested methods

Require: Program P
Require: Test suite TS
Require: Test criterion TC
Ensure: $pseudo$: {pseudo-tested methods in P}

for $m \in P$incovered$(m, TS) \wedge$ ofInterest(m) **do**
  $variants$: {extreme variants of m}
  if returnsValue(m) **then**
    stripBody(m)
    checkReturnType(m)
    $variants \leftarrow$ fixReturnValues(m)
  else
    $variants \leftarrow$ stripBody(m)
  end if
end for
$failure \leftarrow false$
for $v \in variants$ **do**
  $P' \leftarrow$ replace(m, v, P)
  $failure \leftarrow failure \vee$ run(TS, P')
  if $\neg failure$ **then**
    $pseudo \leftarrow pseudo \cup m$
  end if
end forreturn $pseudo$

larger set of study objects and a different tool support for the detection of pseudo-tested methods.

We analyzed each study subject following the procedure described in Section 6.2.2. The results are summarized in Table 6.2. The second column shows the total number of methods excluding constructors. The third, lists the methods covered by the test suite. The following column shows the ratio of covered methods. The *#MUA* column shows the number of methods under analysis. The last two columns give the number of pseudo-tested methods (#PSEUDO) and their ratio to the methods under analysis (PS_RATE).

We have made the first independent replication of Niedermayr et al. [Niedermayr 2016]'s study. Our replication confirms that all Java projects contain pseudo-tested methods, even the very well tested ones. This improves the external validity of this empirical fact. The ratio of pseudo-tested methods with respect to analyzed methods ranged from 1% to 46% in our dataset.

6.2.3.2 Developer's Assessment

Also, we want to know which pseudo-tested methods do developers consider worth an additional testing action. Following our exchange with the developers, we expand the

Table 6.2: Number of methods in each project, number of methods under analysis and number of pseudo-tested methods

Project	#Methods	#Covered	C_RATE	#MUA	#PSEUDO	PS_RATE
authzforce	697	325	47%	291	13	4%
aws-sdk-java	177449	2314	1%	1800	224	12%
commons-cli	237	181	76%	141	2	1%
commons-codec	536	449	84%	426	12	3%
commons-collections	2729	1270	47%	1232	40	3%
commons-io	875	664	76%	641	29	5%
commons-lang	2421	1939	80%	1889	47	2%
flink-core	4133	1886	46%	1814	100	6%
gson	624	499	80%	477	10	2%
jaxen	958	616	64%	569	11	2%
jfreechart	7289	3639	50%	3496	476	14%
jgit	6137	3702	60%	2539	296	12%
joda-time	3374	2783	82%	2526	82	3%
jopt-simple	298	265	89%	256	2	1%
jsoup	1110	844	76%	751	28	4%
sat4j-core	2218	613	28%	585	143	24%
pdfbox	8164	2418	30%	2241	473	21%
scifio	3269	895	27%	158	72	46%
spoon	4470	2976	67%	2938	213	7%
urbanairship	2933	2140	73%	1989	28	1%
xwiki-rendering	5002	2232	45%	2049	239	12%
Total	234923	32650	14%	28808	2540	9%

Table 6.3: The pseudo-tested methods systematically analyzed by the lead developers, through a video call.

Project	Sample size	Worth	Percentage	Time spent (HH:MM)
`authzforce`[5]	13 (100%)	6	46%	29 min
`sat4j-core`[6]	35 (25%)	8	23%	1 hr 38 min
`spoon`[7]	53 (25%)	16	23%	1 hr 14 min
Total	101	30	30%	3 hr 21 min

qualitative analysis to a sample of 101 pseudo-tested methods distributed across three of our study subjects. We consulted developers to characterize the pseudo-tested methods that are worth an additional testing action and the ones that are not worth it.

We contact the development teams directly. We select three projects for which the developers have accepted to discuss with us: `authzforce`, `sat4j-core` and `spoon`. We set up a video call with the head of each development team. The goal of the call is to present and discuss a selection of pseudo-tested methods in approximately 90 minutes. With this discussion, we seek to know which pseudo-tested methods developers consider relevant enough to trigger additional work on the test suite and approximate their ratio on each project.

Table 6.3 shows the projects involved, footnotes with links to the online summary of the interviews, the number of pseudo-tested methods included in the random sample, the number of methods worth an additional testing action and the percentage they represent with respect to the sample. We also show how much time we spent in the discussion.

In a sample of 101 pseudo-tested methods, systematically analyzed by the lead developers of 3 mature projects, 30 methods (30%) were considered worth of additional testing actions. The developer decisions are based on a deep understanding of the application domain and design of the application. This means that it is not reasonable to prescribe the absolute absence (zero) of pseudo-tested methods.

6.2.4 Discussion

To conclude, our replication confirms that all Java projects contain pseudo-tested methods, even the very well tested ones, ranging from 1% to 46% in our dataset. Developers of 3 projects consider that 30% methods were considered worth of additional testing actions.

In the light of these conclusions, the immediate next step in our research agenda is to investigate an automatic test improvement technique targeted towards pseudo-tested methods. This technique shall kill two birds with one stone: improve the adequacy of the test suite for pseudo-tested methods; let the developers focus their efforts on core features and relieve them from the test improvement task.

Descartes has been integrated as default test-criterion in DSpot. It means that, DSpot amplifies the test suite in order to detect more extreme mutant and thus reduce the number of pseudo-testedmethods.

6.3 Study of Test Generation for Repair

Automatic repair aims at fixing bugs automatically. One family of automatic repair is the test suite based. This family uses the test suite as an oracle to know whether or not the bug has been fix. In this transversal contribution, we use test generation process to enhance automatic repair process and one can see it as a test amplification process. As a perspective of this work, we could use DSpot and a specifically designed test-criterion to improve the outcome of test suite based repair techniques.

6.3.1 Problem Statement

The first role of test suites is to verify that the program behaves as expected. However, this was without reckoning with daring researchers, test suites have been used for others purpose such as automated program repair.

Automated program repair holds out the promise of saving debugging costs and patching buggy programs more quickly than humans. Given this great potential, there has been a surge of research on automated program repair in recent years and several different techniques have been proposed ([Goues 2012, Nguyen 2013, Xuan 2016b, Pei 2014, Long 2017]).

Test suite based repair is a widely studied family of techniques among many different techniques proposed. Test suite based repair uses the test suite as oracle to verify whether a patch, obtained using the automated program repair technique, fix the bug or not. To do so, the test suite has at least one test method that fail and others that pass. The goal of test suite based repair is to make all the test methods pass, *i.e.* fixes the bug by making the failing test method pass, and does not break others component by keeping other test methods passing (avoiding regression).

However, test suites are in essence input-output specifications and are therefore typically inadequate for completely specifying the expected behavior. That is to say, that a patch that makes all test methods pass can be still incorrect according to the expected behavior of the program. The patches that are overly specific to the used test suite and fail to generalize to other tests are called *overfitting* patches ([Smith 2015]).

This study is focused on synthesis-based, a category of test suite based repair technique. Synthesis-based techniques first use test execution information to build a repair constraint, and then use a constraint solver to synthesize a patch. Typical examples in this category include SemFix ([Nguyen 2013]), Nopol ([Xuan 2016b]), and Angelix ([Mechtaev 2016]).

Thus, an approach is proposed that try to alleviate overfitting problem for synthesis-based techniques called UnsatGuided. It makes use of automatic test case generation technique to obtain additional tests and try to solve the overfitting problem.

The remainder of this section is organized as follows:

6.3.2 UnsatGuided Technique

6.3.2.1 Definitions

Let us consider the input domain I of a program P. In a typical repair scenario, the program is almost correct. There is a bug that only affects a portion of the input domain, called the "buggy input domain" I_{bug}. We call the rest of the input domain, considered correct, $I_{correct}$. By definition, a patch changes the behaviors of a portion of the input domain. This portion is called I_{patch}.

6.3.2.2 Algorithm

The overfitting problem for synthesis-based repair techniques arises because the repair constraint established using an incomplete test suite is not strong enough to fully express the intended semantics of a program. The idea is to strengthen the initial repair constraint by augmenting the initial test suite with additional automatically generated tests. A stronger repair constraint would guide synthesis-based repair techniques towards better patches, *i.e.* patches that are correct or at least suffer less from overfitting.

UnsatGuided is proposed to overcome this problem, which gradually makes use of the new information provided by each automatically generated test to build a possibly stronger final repair constraint. The key underlying idea is that if the additional repair constraint enforced by an automatically generated test has logical contradictions with the repair constraint established so far, then the generated test is likely to have its input points lying in I_{bug} Such tests are called "bug-exposing test" and are discarded, the others are used to strengthen the repair constraints.

Algorithm 4 describes the approach in detail. It takes as input a buggy program *P* to be repaired, a manually written test suite *TS* which contains some passing tests and at least one failing test, a synthesis-based repair technique $T_{synthesis}$, a time budget *TB* allocated for the execution of $T_{synthesis}$, and finally an automatic test case generation tool T_{auto} which uses a certain kind of automatic test case generation technique T_{reg}. The output of the algorithm is a patch *pt* to the buggy program *P*.

The algorithm directly returns an empty patch if $T_{synthesis}$ generates no patches within the time budget (lines 2-3). In case $T_{synthesis}$ generates an initial patch $pt_{initial}$, the algorithm first conducts a set of initialization steps as follows: it sets the automatically generated test suite *AGTS* to be an empty set (line 5), sets the returned patch *pt* to be the

Algorithm 4 : Algorithm for the Proposed Approach UnsatGuided

Require: A buggy program P and its manually written test suite TS
Require: A synthesis-based repair technique $T_{synthesis}$ and the time budget TB
Require: An automatic test case generation tool T_{auto}
Ensure: A patch pt to the buggy program P

1: $pt_{initial} \leftarrow T_{synthesis}(P, TS, TB)$
2: **if** $pt_{initial} = null$ **then**
3: $\quad pt \leftarrow null$
4: **else**
5: $\quad AGTS \leftarrow \emptyset$
6: $\quad pt \leftarrow pt_{initial}$
7: $\quad TS_{aug} \leftarrow TS$
8: $\quad t_{initial} \leftarrow getPatchGenTime(T_{synthesis}(P, TS, TB))$
9: $\quad \{file_i\}(i = 1, 2, ..., n) \leftarrow getInvolvedFiles(pt_{initial})$
10: $\quad$ **for** $i = 1$ to n **do**
11: $\quad\quad AGTS \leftarrow AGTS \cup T_{auto}(P, file_i)$
12: $\quad$ **end for**
13: $\quad$ **for** $j = 1$ to $|AGTS|$ **do**
14: $\quad\quad t_j \leftarrow AGTS(j)$
15: $\quad\quad TS_{aug} \leftarrow TS_{aug} \cup \{t_j\}$
16: $\quad\quad pt_{intern} \leftarrow T_{synthesis}(P, TS_{aug}, t_{initial} \times 2)$
17: $\quad\quad$ **if** $pt_{intern} \neq null$ **then**
18: $\quad\quad\quad pt \leftarrow pt_{intern}$
19: $\quad\quad$ **else**
20: $\quad\quad\quad TS_{aug} \leftarrow TS_{aug} - \{t_j\}$
21: $\quad\quad$ **end if**
22: $\quad$ **end for**
23: **end ifreturn** pt

initial patch $pt_{initial}$ (line 6), sets the augmented test suite TS_{aug} to be the manually written test suite *TS* (line 7), and gets the time used by $T_{synthesis}$ to generate the initial patch $pt_{initial}$ and sets $t_{initial}$ to be the value (line 8). Algorithm 4 then identifies the set of files $\{file_i\}$(*i*=1, 2,..., *n*) involved in the initial patch $pt_{initial}$ (line 9) and for each identified file, it uses the automatic test case generation tool T_{auto} to generate a set of tests that target behaviors related with the file and adds the generated tests to the automatically generated test suite *AGTS* (lines 10-12).

Next, the algorithm will use the test suite *AGTS* to refine the initial patch $pt_{initial}$. For each test t_j in the test suite *AGTS* (line 14), the algorithm first adds it to the augmented test suite TS_{aug} (line 15) and runs technique $T_{synthesis}$ with test suite TS_{aug} and new time budget $t_{initial} \times 2$ against program *P* (line 16). The new time budget is used to quickly identify tests that can potentially contribute to strengthening the repair constraint, and thus improve the scalability of the approach. Then, if the generated patch pt_{intern} is not an empty patch, the algorithm updates the returned patch *pt* with pt_{intern} (lines 17-18). In other words, the algorithm deems test t_j as a good test that can help improve the repair constraint. Otherwise, test t_j is removed from the augmented test suite TS_{aug} (lines 19-20) as t_j is either a bug-exposing test or it slows down the repair process too much. After the above process has been completed for each test in the test suite *AGTS*, the algorithm finally returns patch *pt* as the desirable patch (line 24).

6.3.3 Evaluation

Defects4J ([Just 2014]) has been selected, a known database of real faults from real-world Java programs, as the experimental benchmark. For the approach UnsatGuided to be implemented, Nopol [Xuan 2016b] has been chosen to represent synthesis-based repair techniques. The automatic test case generation tool used in this study is EvoSuite [Fraser 2011a].

We evaluate the effectiveness of UnsatGuided from two points: its impact on the overfitting issue and correctness of the original patch generated by Nopol.

Table 6.4 displays the experimental results on combining Nopol with UnsatGuided (hereafter referred to as Nopol+UnsatGuided). This table only shows the Defects4J bugs that can be originally repaired by Nopol, and their identifiers are listed in column *Bug ID*.

The test generation results by running EvoSuite are shown in the two columns under the column *Tests*, among which the *#EvoTests* column shows the total number of tests generated by EvoSuite for all seeds and the *#Bug-expo* column shows the number of bug-exposing tests among all of the generated tests.

The results obtained by running just Nopol are shown in the columns under the column *Nopol*. The *Time* column shows the time used by Nopol to generate the initial patch. The *incomplete fix (#failing)* column shows what is the overfitting issue of incomplete fixing for the original Nopol patch. Each cell in this column is of the form X (Y), where X can be "Yes" or "No" and Y is a digit number. The "Yes" and "No" mean that the original Nopol patch has and does not have overfitting issue of incomplete fixing respectively. The digit number in parentheses shows the number of bug-exposing tests on which the original Nopol patch fails. Similarly, the *regression (#failing)* column tells what is the overfitting issue of regression introduction for the original Nopol patch, and each cell in this column is of the same form with the column *incomplete fix (#failing)*. The "Yes" and "No" for this column mean that the original Nopol patch has and does not have overfitting issue of regression introduction respectively. The digit number in parentheses shows the number of normal tests on which the original Nopol patch fails. Finally, the column *correctness* shows whether the original Nopol patch is correct, with "Yes" representing correct and "No" representing incorrect.

This study aims to assess the effectiveness of UnsatGuided. It can be seen from the

Table 6.4: Experimental results with Nopol+UnsatGuided on the Defects4j Repository, only show bugs with test-suite adequate patches by plain Nopol.

	Tests		Nopol				Nopol+UnsatGuided						
	#EvoTests	#Bug-expo	Time (hh:mm)	incomplete fix (#failing)	regression (#failing)	correctness	#Removed	#Removed Bug-expo	Avg #Time (hh:mm)	Change ratio (#unique)	fix completeness change (Avg #Removedinc)	regression change (Avg #Removedreg)	correctness
Chart_1	3012	0	00:02	No (0)	No (0)	NO	0	0	03:00	0/30 (1)	same (0)	same (0)	NO
Chart_5	2931	3	00:01	No (0)	Yes (10)	NO	104	3	01:18	27/30 (27)	same (0)	improve (2.9)	NO
Chart_9	3165	0	00:01	No (0)	No (0)	NO	0	0	01:00	0/30 (1)	same (0)	same (0)	NO
Chart_13	852	0	00:02	No (0)	No (0)	NO	0	0	00:24	30/30 (2)	same (0)	same (0)	NO
Chart_15	3711	0	00:04	No (0)	Yes (4)	NO	5	0	06:48	27/30 (23)	same (0)	improve (2.0)	NO
Chart_17	3246	10	00:01	Yes (10)	No (0)	NO	27	0	00:48	0/30 (1)	same (0)	same (0)	NO
Chart_21	1584	0	00:01	No (0)	Yes (6)	NO	0	0	00:48	30/30 (30)	same (0)	improve (6.0)⋆	NO
Chart_25	441	0	00:01	No (0)	Yes (8)	NO	0	0	00:12	8/30 (6)	same (0)	improve (8.0)⋆	NO
Chart_26	2432	0	00:03	No (0)	Yes (6)	NO	6	0	13:36	10/10 (5)	same (0)	improve (6.0)⋆	NO
Lang_44	3039	13	00:01	No (0)	No (0)	**YES**	13	13	00:48	3/30 (2)	same (0)	same (0)	**YES**
Lang_51	3720	1	00:01	No (0)	No (0)	NO	15	1	01:00	29/30 (2)	same (0)	same (0)	NO
Lang_53	2931	0	00:01	No (0)	No (0)	NO	0	0	00:06	26/30 (18)	same (0)	same (0)	NO
Lang_55	606	0	00:01	No (0)	No (0)	**YES**	1	0	00:12	30/30 (1)	same (0)	same (0)	**YES**
Lang_58	6471	0	00:01	No (0)	Yes (5)	NO	5	0	01:42	0/30 (1)	same (0)	same (0)	NO
Lang_63	1383	1	00:01	No (0)	No (0)	NO	33	1	00:36	27/30 (5)	same (0)	same (0)	NO
Math_7	876	2	00:16	Yes (2)	No (0)	NO	0	0	05:00	2/30 (3)	same (0)	same (0)	NO
Math_24	1327	0	00:15	No (0)	No (0)	NO	25	0	24:06	10/10 (10)	same (0)	same (0)	NO
Math_28	219	0	00:17	No (0)	No (0)	NO	0	0	00:30	0/30 (1)	same (0)	same (0)	NO
Math_33	1749	1	00:13	Yes (1)	No (0)	NO	19	0	10:30	28/30 (8)	same (0)	worse (-2.0)	NO
Math_40	831	71	00:16	Yes (71)	Yes (21)	NO	392	0	07:00	7/30 (8)	same (0)	same (0)	NO
Math_41	1224	0	00:06	No (0)	Yes (41)	NO	35	0	02:00	27/30 (27)	same (0)	improve (35.1)	NO
Math_42	1770	19	00:04	Yes (19)	No (0)	NO	2	0	03:54	24/30 (22)	same (0)	same (0)	NO
Math_50	1107	26	00:11	Yes (21)	Yes (45)	NO	23	1	04:36	28/30 (27)	improve (1.1)	improve (41.0)	NO
Math_57	651	0	00:03	No (0)	No (0)	NO	0	0	00:48	15/30 (4)	same (0)	same (0)	NO
Math_58	228	0	00:06	No (0)	No (0)	NO	7	0	00:20	2/30 (2)	same (0)	same (0)	NO
Math_69	897	0	00:01	No (0)	No (0)	NO	30	0	00:12	30/30 (21)	same (0)	same (0)	NO
Math_71	951	0	00:01	No (0)	Yes (56)	NO	17	0	00:24	25/30 (11)	same (0)	improve (53.0)	NO
Math_73	1035	0	00:01	No (0)	Yes (1)	NO	10	0	00:18	25/30 (24)	same (0)	improve (1)⋆	NO
Math_78	1014	0	00:01	No (0)	Yes (44)	NO	49	0	00:24	28/30 (16)	same (0)	improve (34.9)	NO
Math_80	1356	67	00:01	Yes (49)	No (0)	NO	29	1	00:54	29/30 (27)	worse (-17.9)	same (0)	NO
Math_81	1320	4	00:01	Yes (4)	Yes (35)	NO	30	0	00:24	23/30 (22)	same (0)	improve (35.0)⋆	NO
Math_82	510	0	00:01	No (0)	No (0)	NO	0	0	00:08	0/30 (1)	same (0)	same (0)	NO
Math_84	165	0	00:01	No (0)	No (0)	NO	0	0	00:06	0/30 (1)	same (0)	same (0)	NO
Math_85	798	0	00:01	No (0)	No (0)	NO	32	0	00:12	28/30 (11)	same (0)	same (0)	**YES**
Math_87	1866	14	00:01	Yes (13)	Yes (8)	NO	0	0	00:54	29/30 (29)	worse (-1)	improve (8.0)⋆	NO
Math_88	1890	11	00:01	Yes (11)	No (0)	NO	0	0	00:30	06/30 (7)	same (0)	same (0)	NO
Math_105	1353	7	00:09	Yes (7)	Yes (6)	NO	6	0	04:20	29/30 (30)	same (0)	improve (2.9)	NO
Time_4	2778	5	00:01	Yes (5)	Yes (6)	NO	0	0	00:54	23/30 (23)	improve (0.8)	improve (5.7)	NO
Time_7	1491	0	00:01	No (0)	Yes (11)	NO	12	0	00:54	12/30 (13)	same (0)	worse (-1)	NO
Time_11	1497	5	00:04	Yes (5)	No (0)	NO	7	0	01:36	0/30 (1)	same (0)	same (0)	NO
Time_14	687	0	00:01	No (0)	Yes (3)	NO	1	0	00:18	24/30 (23)	same (0)	improve (2.0)	NO
Time_16	1476	0	00:01	No (0)	Yes (6)	NO	5	0	00:24	1/30 (2)	same (0)	improve (1)	NO

column *Change ratio (#unique)* of Table 6.4 that for the 42 buggy versions that can be initially repaired by Nopol, the patches generated for 34 buggy versions have been changed at least for one seed after running Nopol+UnsatGuided. If we consider all executions (one per seed per buggy version), we obtain a total of 1220 patches with Nopol+UnsatGuided. Among the 1220 patches, 702 patches are different from the original patches generated by running Nopol only. Thus, UnsatGuided can significantly impact the output of the Nopol repair process. We will further investigate the quality difference between the new Nopol+UnsatGuided patches and the original Nopol patches.

The results for alleviating the two kinds of overfitting issues by running Nopol+UnsatGuided are displayed in the columns *fix completeness change (Avg #Removedinc)* and *regression change (Avg#Removedreg)* of Table 6.4.

With regard to alleviating the overfitting issue of incomplete fixing, we can see from the column *fix completeness change (Avg#Removedinc)* that UnsatGuided has an effect on 4 buggy program versions (Math_50, Math_80, Math_87 and Time_4). For all those 4 buggy versions, the original Nopol patch already has the overfitting issue of incomplete fixing. With UnsatGuided, the overfitting issue of incomplete fixing has been alleviated in 2 cases (Math_50, Time_4) and worsened for 2 other cases (Math_80, Math_87). This means UnsatGuided is likely to have a minimal positive impact on alleviating overfitting issue of incomplete fixing and can possibly have a negative impact on it.

In terms of alleviating overfitting issue of regression introduction, we can see from the column *regression change (Avg#Removedreg)* that UnsatGuided has an effect on 18 buggy program versions. Among the 18 original Nopol patches for these 18 buggy program versions, UnsatGuided has alleviated the overfitting issue of regression introduction for 16 patches. In addition, for 6 buggy program versions, the overfitting issue of regression introduction of the original Nopol patch has been completely removed. These 6 cases are indicated with ($\star$) in Table 6.4. Meanwhile, UnsatGuided worsens the overfitting issue of regression introduction for two other original Nopol patches (Math_33 and Time_7). It can possibly happen as even though the repair constraint for input points within $I_{correct}$ has been somewhat strengthened (but not completely correct), yet the solution of the constraint happens to be more convoluted. Overall, with 16 positive versus 2 negative cases, UnsatGuided can be considered as effective in alleviating overfitting issue of regression introduction.

6.3.4 Discussion

To sum up, UnsatGuided can effectively alleviate the overfitting issue of regression introduction (16/19 cases), but has minimal positive impact on reducing the overfitting issue of incomplete fixing.

In this study, we used Evosuite, a state-of-the-art test generation tool. An alternative would be to use a test amplification tool, such as DSpot in order to overcome the overfitting problems.

DSpot could amplify the test suite, while discarded bug-exposing amplified test methods, and improve the constraint around the program repair problem. This can be implemented as a specific test-criterion. Thus, some input-amplification operators are well suited to this task such as literals modifications.

6.4 Conclusion

In this chapter, I exposed 3 transversal contributions that I made thanks to the skills that I developed during my thesis. I worked with my researchers colleagues on side topics yet linked to this thesis.

First, the study of program correctness under runtime perturbation. It highlighted the existence of the correctness attraction phenomenon. This first work could be used jointly with DSpot, by integrating the correctness ratio as a test-criterion in order to strengthen the ability of the test suite to detect more runtime perturbation.

Second, the study of pseudo-tested methods and extreme mutations. It showed the prevalence of pseudo-tested methods, all the tests pass even if the whole behavior, *i.e.* body, of such methods is removed. This extreme mutations has been already implemented in DSpot as a test-criterion in order to reduce the number of pseudo-tested methods.

Third, the study of patch overfitting in test-based repair techniques and an approach to overcome it. In this study, we used a test generation tool but using DSpot with a dedicated and specifically designed test-criterion could give a different outcome.

The next chapter gives the short- and long-term perspectives and concludes this thesis.

CHAPTER 7

Conclusion

Contents

7.1 Contribution Summary

7.1.1 DSpot

The major technical contribution of this thesis is an unit test amplification algorithm, implemented in a mature tool called DSpot. DSpot aims at improving existing test methods with respect to a given test-criterion such as branch coverage. It does this in 3 main steps.

1) It amplifies the input of the original test method by applying specific code transformation on the input part of the test method;

2) It removes existing assertions and generates new ones, based on observations done during execution on the state of the programs. It uses commons getters in Java to do so.

3) It uses a test-criterion to select amplified test methods to keep. For instance, one wants to improve the branch coverage of the test suite. DSpot will keep only amplified test methods that cover branches that were not cover before.

DSpot has been developed in Java, for Java programs. However, the whole technique remains applicable for all programming languages.

All the code of DSpot is available on GitHub: https://github.com/STAMP-project/dspot.git. I personally enliven its community by answering questions on the bug tracker, guiding new contributors and setting up methodologies, such as pull-request based development or integration continue, to keep DSpot as clean as possible.

Following this technical contribution, this thesis presented two large-scale evaluations of DSpot's effectiveness.

7.1.2 Automatic Test Amplification For Mutation Score

For a first evaluation, I used mutation score as test-criterion. DSpot has automatically amplified test suites from open-source projects from GitHub and improved the mutation score.

The generated amplified test methods of DSpot have been proposed to external developers of the projects from GitHub through pull-requests. This has been done in order to have the developers assessing the result of DSpot. Over 19 opened pull-requests, 14 of them have been permanently added to the test suites of these projects. It means that DSpot generated amplified test methods that are valuables for external developers.

Also, I evaluated DSpot by amplifying 40 test classes of heavily tested projects from GitHub, using also the mutation score as test-criterion. This evaluation shows that DSpot is able to generate amplified test methods that increase the mutation score.

7.1.3 Automatic Test Amplification For Behavioral Changes Detection

In a second evaluation, I used DSpot in the context of continuous integration. The goal was to generate amplified test methods that detect behavioral changes.

I took open-source projects from GitHub and a selection of commits. This evaluation showed that DSpot is able to generate amplified test methods that detect 25 behavioral change over 60, which is an achievement. It also highlights the fact that DSpot can be easily implemented in the life cycle of software, like continuous integration.

This evaluation brings evidence that DSpot has to potential to be a concrete part of continuous integration by improving the process of program evolution with amplified test methods that are able to distinguish between versions of the same program.

7.1.4 Transversal Contributions

During this thesis, I developed a wide range of knowledge and skills that allowed me to participated to diversified and transversal contributions.

7.1.4.1 Study of Program Correctness

I devised a protocol, named ATTRACT, to study the programs' correctness under runtime perturbation. Ten subjects have been studied using the PONE perturbations model, *i.e.* adds 1 to integer expressions at runtime. It results in 2,509,012 perturbed executions, which makes it one of the largest perturbation experiments ever made. This large number come from the fact that the protocol explores exhaustively the perturbation space. It allows to generalize the result over integer expressions. From this experimentation, the presence of correctness attraction has been observed. Over all perturbed execution, 66% of them do not break the correctness of the output, which is a important proportion. It means that software are quite reliable according to the PONE perturbations model. For a large proportion, programs are able to recover from small perturbations and produce the correct output, assessed by perfect oracle.

7.1.4.2 Study of Pseudo-tested Methods

We replicated the study of Niedermayr et al. [Niedermayr 2016] and confirmed that all Java projects contain pseudo-tested methods, even the very well tested ones, ranging from 1% to 46% in our dataset. From 3 projects, developers considered that 30% methods were worthy of additional testing actions. Pseudo-tested methods is an important issues since the coverage of the test suite ensures that the code is covered while it is not properly tested. This is misleading and developers might think that the program is well tested while it is not. Using test amplification to resolve this issue and test properly pseudo-tested methods is feasible.

7.1.4.3 Study of Test Generation for Repair

To sum up, UnsatGuided can effectively alleviate the overfitting issue of regression introduction (16/19 cases), but has minimal positive impact on reducing the overfitting issue of incomplete fixing. In this study, we used Evosuite, a state-of-the-art test generation tool. An alternative would be to use a test amplification tool, such as DSpot in order to overcome the overfitting problems.

7.2 Short-term Perspectives

In this section, I introduce short-term perspectives for DSpot.

1) Amplified test methods are based on existing ones. The intuition would be that amplified test methods are easy to read as the seed test method. However, since it is still an automatic process, the amplification can result in difficult to read test methods. A lacking key-feature is to make them prettier, and subsection 7.2.1 introduces a way to do it.

2) An obstacle of test amplification's tool adoption might be the way it must be used. For example, DSpot is usable from command line with an executable jar or with a maven command. Even if we put efforts to make this usage easier, developers might not want to type a command line, that can appear complex. In subsection 7.2.2, I introduce the idea of a web interface for test amplification tools. This web interface would allow users to try test amplification tools, like DSpot, remotely using a graphic interface. The ambitious of this is to spread test amplification approaches over open-source communities. This would have 2 benefits: 1) Users would be welcomed with a friendly interface, and could discover test amplification step-by-step without modifying anything on their own computer. 2) It would collect data on the usage of test amplification and make a large experimentation thanks to users that, on their own, would use the amplification tool.

7.2.1 Prettifying Amplified Test Methods

This section presents an algorithm that would make prettier amplified test methods. Here prettier means that the amplified test method would have less noise, *i.e.* would be easier to read for the developed. In this context, I qualify as noise extra statements that are not required, redundant method calls or meaningless names from the generation process. After applying this approach, the labels, *i.e.* variables names and test method name, would be clear and carry the intent of the variable or of the test method and all extra statement would be removed. This algorithm will take as input a set of amplified test methods, generated by a test amplification tool such as DSpot, and a test-criterion to output a set of prettier amplified test methods. It will ensure that the prettified amplified test methods have the same quality than the input amplified test methods, with respect to the given test-criterion, *e.g.* the mutation score.

This algorithm would work in 3 main steps:

1) it would minimize the number of statement (subsubsection 7.2.1.1);

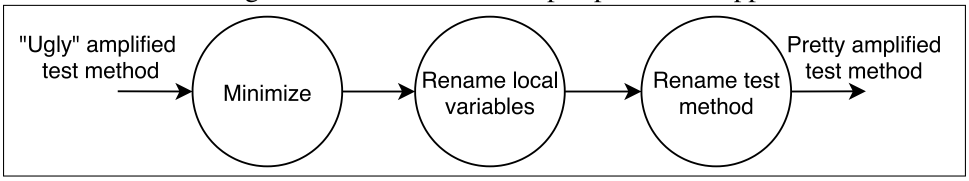
2) it would rename all the local variables with a name according to the context (subsubsection 7.2.1.2);

3) it would rename the amplified test methods according to its body (subsubsection 7.2.1.3).

This workflow is summarize in subsection 7.2.1.

Such algorithm could be evaluated by comparing how the algorithm would minimize an amplified test methods and a minimazation done manually by a developers. Would the algorithm be faster than the human? Would the algorithm obtain a better, a worse or as similar minimization than the human?

Figure 7.1: Overview of DSpot-prettifier's approach.



7.2.1.1 Minimization

The minimization step would aim at reducing the size of the statement and the number statement to the minimal. This would be done in 2 majors steps:

1) modifying the code using static analysis to avoid redundancy or useless local variable declaration; The algorithm will replace multiple method calls, *e.g.* to getters, that are the same by a local variable. In the other way, it will replace local variables that are used only one time.

2) applying a search-based algorithm to remove the maximum number of statements, *w.r.t.* to the specified test-criterion. The intuition is as follow: The algorithm would remove one statement; it tries to compile the new test method; if it fails, it means that this statement is needed to compile the test method and it must keep it; otherwise, it measure the quality of the new test method according to the test criterion; if it remains the same, it can remove the statement definitively, otherwise it cannot be removed; It will repeat this process for all the statement inside the test method, starting by the end of the body.

7.2.1.2 Rename Local Variables

After that the algorithm would have minimize the statements, the next step would be to rename all the local variables. The objectives would be to have variables with clear names, that give hints about the role and the intention of variables.

To do this, I could use Context2Name[1] [Bavishi 2018] which is a deep learning-based approach to infer natural variable names. The idea behind Context2Name is to exploit the usage context, *i.e.* the surrounding lexical tokens, of the variable to infer a proper name in natural languages. For each local variables, the algorithm would use Context2Name to generate a new name for the local variable.

7.2.1.3 Rename Amplified Test Method

The final step would renaming the amplified test method. This could be done in a similar way than renaming local variables subsubsection 7.2.1.2 but at the method level. The goal would be to have a clear name for the amplified test methods that give directly the intention

[1] https://github.com/rbavishi/Context2Name

of the test method. In this way, developers would understand quicker what is the purpose of this test method The major stake would be that developers are more likely to integrate amplified test methods into their test suite.

To do this, I could use Code2Vec[2][Alon 2018] which is a neural network for learning distributed representations of code. However, since Code2Vec would use the whole body the test method, it would be mandatory that the algorithm renames the local variables before the amplified test methods.

7.2.2 Collecting Developers Feedback

The idea would be to provide to users a web interface, on which the user would have simply to put the URL of its GitHub repository. Then, we would retrieve the project and run DSpot on it. This idea is largely inspired by CommitGuru[3][Rosen 2015]. CommitGuru is a tool that identifies and predicts risky software commits. The user gives the URL of its git repository, *e.g.* on github, and CommitGuru analyzes the project and its commits to highlights potential threats.

The goal of this web interface would be to allow new users discover test amplification tools. The users could consult the result of the amplification from the web interface, or by receiving an email. All the data could be accessed from the web interface in order to allow the reproduction of test amplifications.

A UI mockup of the web interface is shown in Figure 7.2. On this picture, one can see that the users would have only to put the url of its repository and its email, in order to receive the result for example. At the bottom, on can see a list of projects that used recently the test amplification tool.

7.3 Long-term Perspectives

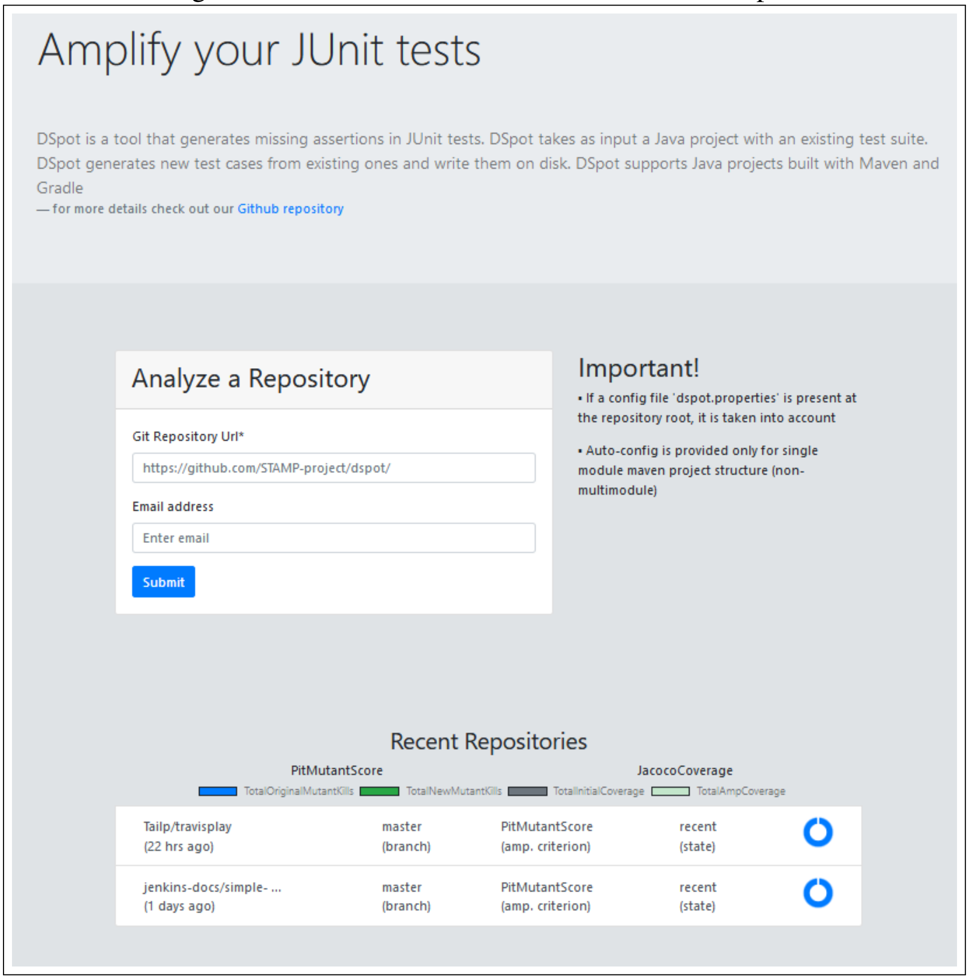
In my vision, there are 2 long-term perspectives for DSpot:

1) sometimes, test-improvement resulting from a given test class should not be in the same class, *e.g.* subsubsection 4.3.1.8. This might be a limitation on the adoption of test amplification tools by industrials since they might be confused by the fact that the component tested is not any more related to the original test class. **Would we able to find the best location for a given amplified test methods?** This new location would take into account 2 aspect: 1) objects used in the test and the methods called on then, *i.e.* the input part. 2) the values that are asserted, *i.e.* the oracle part.

2) Inspired from Repairnator [Urli 2018], which is bot that automatically executes test-based repairs programs to fix CI build failures. Repairnator crawls builds status on Travis,

[2] `https://github.com/tech-srl/code2vec`
[3] `http://commit.guru/`

Figure 7.2: Screenshot of the web interface for DSpot.



a continuous integration service on GitHub. Then, Repairnator launches automatic repair tools to failing builds and then, it is able to propose the patch throught a pull-request on the GitHub repository. **Would we able to automatically amplify test suite from the CI and provide an test method that a developer did not?** In the same way, we could imagine a bot that uses test amplification tools. For example, the bot would launch DSpot on passing build, a contrario than Repairnator is executed on failing build, to amplify the test suite. The goal of this amplification would be even to make evolve the test suite using the mutation score as test-criterion such as shown in Section 4.1 or to detect a regression such as shown in subsection 5.1.4.

Bibliography

[Abdulla 2004] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson and Mayank Saksena. *A Survey of Regular Model Checking*. In Philippa Gardner and Nobuko Yoshida, editors, CONCUR 2004 - Concurrency Theory, pages 35–48, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. (Cited on page 1.)

[Alon 2018] Uri Alon, Meital Zilberstein, Omer Levy and Eran Yahav. *code2vec: Learning Distributed Representations of Code*. CoRR, vol. abs/1803.09473, 2018. (Cited on page 124.)

[Apiwattanapong 2006] Taweesup Apiwattanapong, Raul Santelices, Pavan Kumar Chittimalli, Alessandro Orso and Mary Jean Harrold. *Matrix: Maintenance-oriented testing requirements identifier and examiner*. In Testing: Academic and Industrial Conference-Practice And Research Techniques, 2006. TAIC PART 2006. Proceedings, pages 137–146. IEEE, 2006. (Cited on pages 18, 21, 22 and 29.)

[Baudry 2002] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel and Yves Le Traon. *Automatic Test Cases Optimization Using a Bacteriological Adaptation Model: Application to .NET Components*. In Proceedings of the 17th IEEE International Conference on Automated Software Engineering, ASE '02, pages 253–, Washington, DC, USA, 2002. IEEE Computer Society. (Cited on page 29.)

[Baudry 2005a] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel and Yves Le Traon. *Automatic Test Cases Optimization: a Bacteriologic Algorithm*. IEEE Software, vol. 22, no. 2, pages 76–82, March 2005. (Cited on pages 11, 15 and 29.)

[Baudry 2005b] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel and Le Traon Yves. *From Genetic to Bacteriological Algorithms for Mutation-Based Testing*. Software, Testing, Verification & Reliability journal (STVR), vol. 15, no. 2, pages 73–96, June 2005. (Cited on pages 11, 15 and 29.)

[Baudry 2006] Benoit Baudry, Franck Fleurey and Yves Le Traon. *Improving Test Suites for Efficient Fault Localization*. In Proceedings of the 28th International Conference on Software Engineering, ICSE '06, pages 82–91, 2006. (Cited on pages 14, 16, 29 and 32.)

[Bavishi 2018] Rohan Bavishi, Michael Pradel and Koushik Sen. *Context2Name: A Deep Learning-Based Approach to Infer Natural Variable Names from Usage Contexts*. CoRR, vol. abs/1809.05193, 2018. (Cited on page 123.)

[Beck 2003] K. Beck. Test-driven development: by example. Addison-Wesley Professional, 2003. (Cited on page 1.)

[Beller 2015a] Moritz Beller, Georgios Gousios, Annibale Panichella and Andy Zaidman. *When, how, and why developers (do not) test in their IDEs*. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), pages 179–190. ACM, 2015. (Cited on pages 1 and 8.)

[Beller 2015b] Moritz Beller, Georgios Gousios and Andy Zaidman. *How (Much) Do Developers Test?* In Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE), pages 559–562. IEEE Computer Society, 2015. (Cited on pages 1 and 8.)

[Beller 2017] Moritz Beller, Georgios Gousios and Andy Zaidman. *TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration*. In Proceedings of the 14th working conference on mining software repositories, 2017. (Cited on page 50.)

[Beller 2019] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann and Andy Zaidman. *Developer Testing in The IDE: Patterns, Beliefs, And Behavior*. IEEE Transactions on Software Engineering, vol. 45, no. 3, pages 261–284, 2019. (Cited on pages 1 and 8.)

[Blackburn 2006] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage and Ben Wiedermann. *The DaCapo Benchmarks: Java Benchmarking Development and Analysis*. In Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM. (Cited on page 24.)

[Bloem 2014] Roderick Bloem, Robert Koenighofer, Franz Röck and Michael Tautschnig. *Automating test-suite augmentation*. In Quality Software (QSIC), 2014 14th International Conference on, pages 67–72. IEEE, 2014. (Cited on pages 12, 16 and 29.)

[Böhme 2013] Marcel Böhme, Bruno C d S Oliveira and Abhik Roychoudhury. *Regression tests to expose change interaction errors*. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pages 334–344. ACM, 2013. (Cited on pages 20, 21 and 29.)

[Böhme 2014] Marcel Böhme and Abhik Roychoudhury. *Corebench: Studying complexity of regression errors*. In Proceedings of the 2014 International Symposium on Software Testing and Analysis, pages 105–115. ACM, 2014. (Cited on pages 21 and 32.)

[Brereton 2007] Pearl Brereton, Barbara A. Kitchenham, David Budgen, Mark Turner and Mohamed Khalil. *Lessons from applying the systematic literature review process within the software engineering domain*. Journal of Systems and Software, vol. 80, no. 4, pages 571–583, 2007. (Cited on page 9.)

[Carzaniga 2014] Antonio Carzaniga, Alberto Goffi, Alessandra Gorla, Andrea Mattavelli and Mauro Pezzè. *Cross-checking Oracles from Intrinsic Software Redundancy*. In Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pages 931–942, 2014. (Cited on pages 25, 28, 29 and 32.)

[Coles 2016] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis and Anthony Ventresque. *PIT: A Practical Mutation Testing Tool for Java (Demo)*. In Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, pages 449–452, New York, NY, USA, 2016. ACM. (Cited on pages 48 and 108.)

[Cooper 1998] Harris M Cooper. Synthesizing research: A guide for literature reviews, volume 2. Sage, 1998. (Cited on page 9.)

[Cornu 2015] Benoit Cornu, Lionel Seinturier and Martin Monperrus. *Exception handling analysis and transformation using fault injection: Study of resilience against unanticipated exceptions*. Information and Software Technology, vol. 57, pages 66–76, 2015. (Cited on pages 23, 24 and 29.)

[Dallmeier 2010] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack and Andreas Zeller. *Generating Test Cases for Specification Mining*. In Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA '10, pages 85–96, New York, NY, USA, 2010. ACM. (Cited on pages 25, 28, 29 and 32.)

[Danglot 2018] Benjamin Danglot, Philippe Preux, Benoit Baudry and Martin Monperrus. *Correctness attraction: a study of stability of software behavior under runtime perturbation*. Empirical Software Engineering, vol. 23, no. 4, pages 2086–2119, Aug 2018. (Cited on pages 99, 102 and 105.)

[Danglot 2019a] Benjamin Danglot, Martin Monperrus, Walter Rudametkin and Benoit Baudry. *An Approach and Benchmark to Detect Behavioral Changes of Commits in Continuous Integration*. CoRR, vol. abs/1902.08482, 2019. (Cited on page 72.)

[Danglot 2019b] Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus and Benoit Baudry. *A Snowballing Literature Study on Test Amplification*. Journal of Systems and Software, page 110398, 2019. (Cited on pages 4, 8 and 35.)

[Danglot 2019c] Benjamin Danglot, Oscar Luis Vera-Pérez, Benoit Baudry and Martin Monperrus. *Automatic test improvement with DSpot: a study with ten mature open-source projects*. Empirical Software Engineering, Apr 2019. (Cited on pages 48 and 77.)

[Daniel 2009a] B. Daniel, V. Jagannath, D. Dig and D. Marinov. *ReAssert: Suggesting Repairs for Broken Unit Tests*. In 2009 IEEE/ACM International Conference on Automated Software Engineering, pages 433–444, Nov 2009. (Cited on pages 74 and 99.)

[Daniel 2009b] Brett Daniel, Vilas Jagannath, Danny Dig and Darko Marinov. *ReAssert: Suggesting Repairs for Broken Unit Tests*. In 2009 IEEE/ACM International Conference on Automated Software Engineering, pages 433–444, 2009. (Cited on pages 26, 28, 29 and 32.)

[Dijkstra 1989] Edsger Dijkstra. *On the cruelty of really teaching computing science*. Communications of The ACM - CACM, vol. 32, 01 1989. (Cited on page 1.)

[D'Silva 2008] V. D'Silva, D. Kroening and G. Weissenbacher. *A Survey of Automated Techniques for Formal Software Verification*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 27, no. 7, pages 1165–1178, July 2008. (Cited on page 1.)

[Duvall 2007] Paul M Duvall, Steve Matyas and Andrew Glover. Continuous integration: improving software quality and reducing risk. Pearson Education, 2007. (Cited on page 71.)

[Edvardsson 2002] Jon Edvardsson. *A Survey on Automatic Test Data Generation*. 03 2002. (Cited on page 35.)

[Falleri 2014] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez and Martin Monperrus. *Fine-grained and Accurate Source Code Differencing*. In Proceedings of the International Conference on Automated Software Engineering, pages 313–324, 2014. (Cited on page 76.)

[Fang 2015] Lu Fang, Liang Dou and Guoqing Xu. *PERFBLOWER: Quickly Detecting Memory-Related Performance Problems via Amplification*. In LIPIcs-Leibniz

International Proceedings in Informatics, volume 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015. (Cited on pages 23, 24, 29 and 32.)

[Fowler 2006] Martin Fowler and Matthew Foemmel. *Continuous integration*. ThoughtWorks https://www.thoughtworks.com/continuous-integration, vol. 122, page 14, 2006. (Cited on page 71.)

[Fraser 2011a] Gordon Fraser and Andrea Arcuri. *EvoSuite: automatic test suite generation for object-oriented software*. In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM. (Cited on pages 2 and 115.)

[Fraser 2011b] Gordon Fraser and Andrea Arcuri. *Evosuite: automatic test suite generation for object-oriented software*. In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pages 416–419. ACM, 2011. (Cited on pages 24 and 32.)

[Fraser 2011c] Gordon Fraser and Andreas Zeller. *Generating parameterized unit tests*. In Proceedings of the 2011 International Symposium on Software Testing and Analysis, pages 364–374. ACM, 2011. (Cited on pages 14, 16 and 29.)

[Fraser 2015] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri and Frank Padberg. *Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study*. ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 24, no. 4, page 23, 2015. (Cited on page 2.)

[Goues 2012] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest and Westley Weimer. *GenProg: A Generic Method for Automatic Software Repair*. IEEE Trans. Software Eng., vol. 38, no. 1, pages 54–72, 2012. (Cited on page 112.)

[Hamlet 1993] Dick Hamlet and Jeff Voas. *Faults on its sleeve: amplifying software reliability testing*. ACM SIGSOFT Software Engineeri ng Notes, vol. 18, no. 3, pages 89–98, 1993. (Cited on pages 9, 10, 25, 28 and 29.)

[Harder 2003] Michael Harder, Jeff Mellen and Michael D. Ernst. *Improving Test Suites via Operational Abstraction*. In Proc. of the Int. Conf. on Software Engineering (ICSE), pages 60–71, 2003. (Cited on pages 13, 16 and 29.)

[Hetzel 1988] William C Hetzel. The complete guide to software testing. QED Information Sciences, Inc., Wellesley, MA, USA, 2nd édition, 1988. (Cited on page 8.)

[Hilton 2016] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov and Danny Dig. *Usage, Costs, and Benefits of Continuous Integration in Open-source Projects*. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, pages 426–437, New York, NY, USA, 2016. ACM. (Cited on page 71.)

[Hilton 2018a] Michael Hilton, Jonathan Bell and Darko Marinov. *A large-scale study of test coverage evolution*. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE), pages 53–63. ACM, 2018. (Cited on page 8.)

[Hilton 2018b] Michael Hilton, Jonathan Bell and Darko Marinov. *A Large-scale Study of Test Coverage Evolution*. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, pages 53–63, New York, NY, USA, 2018. ACM. (Cited on page 77.)

[Hutchins 1994] Monica Hutchins, Herb Foster, Tarak Goradia and Thomas Ostrand. *Experiments of the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria*. In Proceedings of the 16th International Conference on Software Engineering, ICSE '94, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. (Cited on page 18.)

[Jalali 2012] Samireh Jalali and Claes Wohlin. *Systematic literature studies: database searches vs. backward snowballing*. In Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement, pages 29–38. ACM, 2012. (Cited on pages 9 and 10.)

[Joshi 2007] Pallavi Joshi, Koushik Sen and Mark Shlimovich. *Predictive Testing: Amplifying the Effectiveness of Software Testing*. In Proc. of the ESEC/FSE: Companion Papers, ESEC-FSE companion '07, pages 561–564, New York, NY, USA, 2007. ACM. (Cited on pages 9, 10, 26, 28 and 29.)

[Just 2014] René Just, Darioush Jalali and Michael D. Ernst. *Defects4J: A database of existing faults to enable controlled testing studies for Java programs*. In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), pages 437–440, San Jose, CA, USA, July 23–25 2014. (Cited on page 115.)

[Kitchenham 2004] Barbara Kitchenham. *Procedures for performing systematic reviews*. Technical report, Keele University, 2004. (Cited on page 9.)

[Leung 2012] Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh Gupta, Ranjit Jhala and Sorin Lerner. *Verifying GPU kernels by test amplification*. ACM SIGPLAN Notices, vol. 47, no. 6, pages 383–394, 2012. (Cited on pages 9, 10, 23, 24 and 29.)

[Li 2007] X. Li and D. Yeung. *Application-Level Correctness and its Impact on Fault Tolerance*. In 2007 IEEE 13th International Symposium on High Performance Computer Architecture, pages 181–192, Feb 2007. (Cited on page 102.)

[Long 2017] Fan Long, Peter Amidon and Martin Rinard. *Automatic inference of code transforms for patch generation*. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pages 727–739. ACM, 2017. (Cited on page 112.)

[Madeyski 2010] Lech Madeyski. Test-driven development: An empirical evaluation of agile practice. Springer, 2010. (Cited on page 8.)

[Marinescu 2013] Paul Dan Marinescu and Cristian Cadar. *KATCH: high-coverage testing of software patches*. page 235. ACM Press, 2013. (Cited on pages 20, 21, 22 and 29.)

[Marri 2010] Madhuri R Marri, Suresh Thummalapenta, Tao Xie, Nikolai Tillmann and Jonathan de Halleux. *Retrofitting unit tests for parameterized unit testing*. Technical report, North Carolina State University, 2010. (Cited on pages 11, 16, 29 and 32.)

[McMinn 2004] Phil McMinn. *Search-based software test data generation: a survey*. Software Testing, Verification and Reliability, vol. 14, no. 2, pages 105–156, 2004. (Cited on page 35.)

[Mechtaev 2016] Sergey Mechtaev, Jooyong Yi and Abhik Roychoudhury. *Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis*. In Proceedings of the 38th International Conference on Software Engineering, ICSE '16, pages 691–701, New York, NY, USA, 2016. ACM. (Cited on page 112.)

[Milani Fard 2014] Amin Milani Fard, Mehdi Mirzaaghaei and Ali Mesbah. *Leveraging existing tests in automated test generation for web applications*. In Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, pages 67–78. ACM, 2014. (Cited on pages 13, 16, 29 and 32.)

[Mirzaaghaei 2012] Mehdi Mirzaaghaei, Fabrizio Pastore and Mauro Pezze. *Supporting test suite evolution through test case adaptation*. In Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on, pages 231–240. IEEE, 2012. (Cited on pages 20 and 29.)

[Mirzaaghaei 2014] Mehdi Mirzaaghaei, Fabrizio Pastore and Mauro Pezzè. *Automatic test case evolution*. Software Testing, Verification and Reliability, vol. 24, no. 5, pages 386–411, 2014. (Cited on pages 20 and 29.)

[Mouelhi 2009] Tejeddine Mouelhi, Yves Le Traon and Benoit Baudry. *Transforming and selecting functional test cases for security policy testing*. In Software Testing Verification and Validation, 2009. ICST'09. International Conference on, pages 171–180. IEEE, 2009. (Cited on pages 26, 28 and 29.)

[Nguyen 2013] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury and Satish Chandra. *SemFix: Program Repair via Semantic Analysis*. In Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press. (Cited on page 112.)

[Niedermayr 2016] Rainer Niedermayr, Elmar Juergens and Stefan Wagner. *Will my tests tell me if I break this code?* In Proceedings of the International Workshop on Continuous Software Evolution and Delivery, pages 23–29, New York, NY, USA, 2016. ACM Press. (Cited on pages 107, 108, 109 and 121.)

[Ostrand 1988] Thomas J. Ostrand and Marc J. Balcer. *The category-partition method for specifying and generating fuctional tests*. Communications of the ACM, vol. 31, no. 6, pages 676–686, 1988. (Cited on page 25.)

[Pacheco 2005a] Carlos Pacheco and Michael D. Ernst. *Eclat: Automatic generation and classification of test inputs*. In ECOOP 2005 — Object-Oriented Programming, 19th European Conference, pages 504–527, Glasgow, Scotland, July 2005. (Cited on page 2.)

[Pacheco 2005b] Carlos Pacheco and Michael D Ernst. *Eclat: Automatic generation and classification of test inputs*. In Proceedings of the 19th European conference on Object-Oriented Programming, pages 504–527, Berlin, Heidelberg, 2005. Springer-Verlag, Springer Berlin Heidelberg. (Cited on pages 14, 16, 29 and 32.)

[Palikareva 2016] Hristina Palikareva, Tomasz Kuchta and Cristian Cadar. *Shadow of a doubt: testing for divergences between software versions*. In Proceedings of the 38th International Conference on Software Engineering, pages 1181–1192. ACM, 2016. (Cited on pages 21, 22, 29 and 32.)

[Palomb] Fabio Palomb and Andy Zaidman. *The Smell of Fear: On the Relation between Test Smells and Flaky Tests*. Empirical Software Engineering (EMSE). To Appear. (Cited on page 22.)

[Palomba 2017] Fabio Palomba and Andy Zaidman. *Does Refactoring of Test Smells Induce Fixing Flaky Tests?* In 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 1–12. IEEE Computer Society, 2017. (Cited on page 22.)

[Patrick 2017] Matthew Patrick and Yue Jia. *KD-ART: Should we intensify or diversify tests to kill mutants?* Information and Software Technology, vol. 81, pages 36–51, 2017. (Cited on pages 12, 16 and 29.)

[Pawlak 2015] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera and Lionel Seinturier. *Spoon: A Library for Implementing Analyses and Transformations of Java Source Code*. Software: Practice and Experience, vol. 46, pages 1155–1179, 2015. (Cited on pages 45 and 103.)

[Pei 2014] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer and A. Zeller. *Automated Fixing of Programs with Contracts*. IEEE Transactions on Software Engineering, vol. 40, no. 5, pages 427–449, May 2014. (Cited on page 112.)

[Petersen 2008] Kai Petersen, Robert Feldt, Shahid Mujtaba and Michael Mattsson. *Systematic Mapping Studies in Software Engineering*. In EASE, volume 8, pages 68–77, 2008. (Cited on pages 9 and 10.)

[Pezze 2013] Mauro Pezze, Konstantin Rubinov and Jochen Wuttke. *Generating effective integration test cases from unit ones*. In Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on, pages 11–20. IEEE, 2013. (Cited on pages 13, 29 and 32.)

[Qi 2010] Dawei Qi, Abhik Roychoudhury and Zhenkai Liang. *Test generation to expose changes in evolving programs*. In Proceedings of the IEEE/ACM international conference on Automated software engineering, pages 397–406, 2010. (Cited on pages 19, 22 and 29.)

[Rinard 2005] Martin Rinard, Cristian Cadar and Huu Hai Nguyen. *Exploring the Acceptability Envelope*. In Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '05, pages 21–30, New York, NY, USA, 2005. ACM. (Cited on page 102.)

[Röβler 2012] Jeremias Röβler, Gordon Fraser, Andreas Zeller and Alessandro Orso. *Isolating failure causes through test case generation*. In Proceedings of the 2012 International Symposium on Software Testing and Analysis, pages 309–319. ACM, 2012. (Cited on pages 14, 16, 29 and 32.)

[Roche 2013] James Roche. *Adopting DevOps Practices in Quality Assurance*. Commun. ACM, vol. 56, 2013. (Cited on page 1.)

[Rojas 2016] José Miguel Rojas, Gordon Fraser and Andrea Arcuri. *Seeding strategies in search-based unit test generation*. Software Testing, Verification and Reliability, vol. 26, no. 5, pages 366–401, 2016. (Cited on pages 12 and 29.)

[Rosen 2015] Christoffer Rosen, Ben Grawi and Emad Shihab. *Commit Guru: Analytics and Risk Prediction of Software Commits*. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pages 966–969, New York, NY, USA, 2015. ACM. (Cited on page 124.)

[Saff 2004] David Saff and Michael D Ernst. *An experimental evaluation of continuous testing during development*. In ACM SIGSOFT Software Engineering Notes, volume 29, pages 76–85. ACM, 2004. (Cited on page 74.)

[Santelices 2008] Raul Santelices, Pavan Kumar Chittimalli, Taweesup Apiwattanapong, Alessandro Orso and Mary Jean Harrold. *Test-suite augmentation for evolving software*. In 23rd IEEE/ACM International Conference on, pages 218–227. IEEE, 2008. (Cited on pages 18, 22 and 29.)

[Santelices 2011] Raul Santelices and Mary Jean Harrold. *Applying aggressive propagation-based strategies for testing changes*. In IEEE Fourth International Conference on Software Testing, Verification and Validation, pages 11–20. IEEE, 2011. (Cited on pages 18, 22 and 29.)

[SIR] *Software-artifact Infrastructure Repository*. `http://sir.unl.edu`. Accessed: 2017-05-17. (Cited on pages 17 and 32.)

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

[Tillmann 2006] Nikolai Tillmann and Wolfram Schulte. *Unit tests reloaded: Parameterized unit testing with symbolic execution*. IEEE software, vol. 23, no. 4, pages 38–47, 2006. (Cited on pages 11, 16 and 29.)

[Tonella 2004] Paolo Tonella. *Evolutionary Testing of Classes*. In Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '04, pages 119–128, New York, NY, USA, 2004. ACM. (Cited on pages 38, 45 and 49.)

[Urli 2018] Simon Urli, Zhongxing Yu, Lionel Seinturier and Martin Monperrus. *How to Design a Program Repair Bot? Insights from the Repairnator Project*. In 40th International Conference on Software Engineering, Track Software Engineering in Practice, pages 95–104, 2018. (Cited on page 124.)

[Vera-Pérez 2018a] Oscar L. Vera-Pérez, Martin Monperrus and Benoit Baudry. *Descartes: A PITest Engine to Detect Pseudo-Tested Methods*. In Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), pages 908–911, 2018. (Cited on page 108.)

[Vera-Pérez 2018b] Oscar Luis Vera-Pérez, Benjamin Danglot, Martin Monperrus and Benoit Baudry. *A comprehensive study of pseudo-tested methods*. Empirical Software Engineering, Sep 2018. (Cited on pages 77, 99 and 102.)

[Voas 1995] Jeffrey M. Voas and Keith W Miller. *Software testability: The new verification*. IEEE software, vol. 12, no. 3, pages 17–28, 1995. (Cited on page 83.)

[Wang 2014] Haijun Wang, Xiaohong Guan, Qinghua Zheng, Ting Liu, Chao Shen and Zijiang Yang. *Directed test suite augmentation via exploiting program dependency*. In Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis, pages 1–6. ACM, 2014. (Cited on pages 19, 21, 22 and 29.)

[Wohlin 2014] Claes Wohlin. *Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering*. In Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, page 38. ACM, 2014. (Cited on page 9.)

[Xie 2006] Tao Xie. *Augmenting Automatically Generated Unit-test Suites with Regression Oracle Checking*. In Proceedings of the 20th European Conference on Object-Oriented Programming, pages 380–403, 2006. (Cited on pages 25, 28, 29, 39, 45, 49 and 70.)

[Xu 2009] Zhihong Xu and Gregg Rothermel. *Directed test suite augmentation*. In Software Engineering Conference, 2009. APSEC'09. Asia-Pacific, pages 406–413. IEEE, 2009. (Cited on pages 16, 22 and 29.)

[Xu 2010a] Zhihong Xu, Myra B Cohen and Gregg Rothermel. *Factors affecting the use of genetic algorithms in test suite augmentation*. In Proceedings of the 12th annual conference on Genetic and evolutionary computation, pages 1365–1372. ACM, 2010. (Cited on pages 17 and 29.)

[Xu 2010b] Zhihong Xu, Yunho Kim, Moonzoo Kim, Gregg Rothermel and Myra B Cohen. *Directed test suite augmentation: techniques and tradeoffs*. In Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering, pages 257–266. ACM, 2010. (Cited on pages 17 and 29.)

[Xu 2011] Zhihong Xu, Yunho Kim, Moonzoo Kim and Gregg Rothermel. *A hybrid directed test suite augmentation technique*. In Software Reliability Engineering (IS-

SRE), 2011 IEEE 22nd International Symposium on, pages 150–159. IEEE, 2011. (Cited on pages 17, 21, 22 and 29.)

[Xu 2015] Zhihong Xu, Yunho Kim, Moonzoo Kim, Myra B Cohen and Gregg Rothermel. *Directed test suite augmentation: an empirical investigation*. Software Testing, Verification and Reliability, vol. 25, no. 2, pages 77–114, 2015. (Cited on pages 18, 21, 22 and 29.)

[Xuan 2014] Jifeng Xuan and Martin Monperrus. *Test case purification for improving fault localization*. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 52–63. ACM, 2014. (Cited on pages 27, 28 and 29.)

[Xuan 2015] Jifeng Xuan, Xiaoyuan Xie and Martin Monperrus. *Crash Reproduction via Test Case Mutation: Let Existing Test Cases Help*. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pages 910–913, New York, NY, USA, 2015. ACM. (Cited on pages 15, 16 and 29.)

[Xuan 2016a] Jifeng Xuan, Benoit Cornu, Matias Martinez, Benoit Baudry, Lionel Seinturier and Martin Monperrus. *B-Refactoring: Automatic Test Code Refactoring to Improve Dynamic Analysis*. Information and Software Technology, vol. 76, pages 65–80, 2016. (Cited on pages 27, 28, 29 and 32.)

[Xuan 2016b] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas, Thomas Durieux, Daniel Le Berre and Martin Monperrus. *Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs*. IEEE Transactions on Software Engineering, 2016. (Cited on pages 112 and 115.)

[Xuan 2017] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre and Martin Monperrus. *Nopol: Automatic repair of conditional statement bugs in java programs*. IEEE Transactions on Software Engineering, vol. 43, no. 1, pages 34–55, 2017. (Cited on pages 27 and 32.)

[Yoo 2012] Shin Yoo and Mark Harman. *Test data regeneration: generating new test data from existing test data*. Software Testing, Verification and Reliability, vol. 22, no. 3, pages 171–201, 2012. (Cited on pages 11, 16, 29 and 35.)

[Yoshida 2016] Hiroaki Yoshida, Susumu Tokumoto, Mukul R Prasad, Indradeep Ghosh and Tadahiro Uehara. *FSX: Fine-grained Incremental Unit Test Generation for C/C++ Programs*. In Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, 2016. (Cited on pages 13, 15 and 29.)

[Yu 2013] Zhongxing Yu, Chenggang Bai and Kai-Yuan Cai. *Mutation-oriented Test Data Augmentation for GUI Software Fault Localization*. Inf. Softw. Technol., vol. 55, no. 12, pages 2076–2098, December 2013. (Cited on pages 15 and 29.)

[Yu 2019] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux and Martin Monperrus. *Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the Nopol repair system*. Empirical Software Engineering, vol. 24, no. 1, pages 33–67, Feb 2019. (Cited on page 102.)

[Zaidman 2008] Andy Zaidman, Bart Van Rompaey, Serge Demeyer and Arie van Deursen. *Mining Software Repositories to Study Co-Evolution of Production & Test Code*. In First International Conference on Software Testing, Verification, and Validation (ICST), pages 220–229. IEEE Computer Society, 2008. (Cited on pages 8 and 16.)

[Zaidman 2011] Andy Zaidman, Bart Van Rompaey, Arie van Deursen and Serge Demeyer. *Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining*. Empirical Software Engineering, vol. 16, no. 3, pages 325–364, 2011. (Cited on pages 8 and 16.)

[Zhang 2012] Pingyu Zhang and Sebastian Elbaum. *Amplifying tests to validate exception handling code*. In Proc. of Int. Conf. on Software Engineering (ICSE), pages 595–605. IEEE Press, 2012. (Cited on pages 9, 10, 23, 24 and 29.)

[Zhang 2014] Pingyu Zhang and Sebastian G. Elbaum. *Amplifying Tests to Validate Exception Handling Code: An Extended Study in the Mobile Application Domain*. ACM Trans. Softw. Eng. Methodol., vol. 23, no. 4, pages 32:1–32:28, 2014. (Cited on pages 23, 24 and 29.)

[Zhang 2016] Jie Zhang, Yiling Lou, Lingming Zhang, Dan Hao, Lu Zhang and Hong Mei. *Isomorphic Regression Testing: Executing Uncovered Branches Without Test Augmentation*. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, pages 883–894, New York, NY, USA, 2016. ACM. (Cited on pages 24, 29 and 32.)