

Behavioural Change Detection

PHD THESIS

to obtain the title of

PhD of Science

Specialty : COMPUTER SCIENCE

Defended on **xx**

Benjamin DANGLOT

prepared at Inria Lille-Nord Europe, SPIRALS Team

Thesis committee:

Supervisors: Martin MONPERRUS - KTH Royal Institute of Technology
Lionel SEINTURIER - University of Lille

Reviewers: -

Examiner: -

Chair: -

Invited: Benoit BAUDRY - KTH Royal Institute of Technology
Vincent MASSOL - XWiki

“O sed fugit interea fugit irreparabile tempus audeamus nunc.”

– Rilés

TODO

Acknowledgements

write it

Abstract

write it

Publications

write it

Résumé

write it

Contents

List of Figures	xv
List of Tables	xviii
1 Introduction	1
1.1 STAMP-project	2
1.1.1 Research Directions	3
1.1.2 STAMP's Output	4
1.2 Thesis Vision	5
1.3 Thesis Roadmap	5
1.4 Software and Impact On The Community	6
1.5 Publications	7
2 State of the Art	9
2.1 Introduction	9
2.2 Method	10
2.2.1 Definition	11
2.2.2 Methodology	11
2.2.3 Novelty	13
3 DSpot: A Test Amplification Tool	15
3.1 Definitions	16
3.2 Overview	17
3.2.1 Principle	17
3.2.2 Input & Output	17
3.2.3 Workflow	18
3.2.4 Test method example	19
3.3 Algorithm	20
3.3.1 Input Space Exploration Algorithm	20
3.3.2 Assertion Improvement Algorithm	21
3.3.3 Pseudo-algorithm	22
3.3.4 Flaky tests elimination	23
3.4 Implementation	24
3.4.1 Ecosystem	24
3.5 Conclusion	24

4 Test Amplification For Artificial Behavioral Changes Detection	27
4.1 Introduction	28
4.1.1 Test-criterion: mutation score	28
4.1.2 Mutation score versus coverage	29
4.2 Experimental Protocol	29
4.2.1 Dataset	30
4.2.2 Test Case Selection Process	30
4.2.3 Metrics	32
4.2.4 Methodology	32
4.3 Experimental Results	33
4.3.1 Answer to RQ1	33
4.3.2 Answer to RQ2	43
4.3.3 Answer to RQ3	46
4.3.4 Answer to RQ4	48
4.4 Threats to Validity	49
4.5 Conclusion	49
5 Test Amplification For Commit Behavioral Changes Detection	51
5.1 Introduction	52
5.1.1 Collaborative software development	52
5.1.2 Goal	53
5.2 Motivation & Background	53
5.2.1 Motivating Example	53
5.2.2 Practicability	54
5.2.3 Behavioral Change	55
5.2.4 Behavioral Change Detection	55
5.3 Behavioral Change Detection Approach	55
5.3.1 Overview of DCI	55
5.3.2 Test Selection and Diff Coverage	56
5.3.3 Test Amplification	56
5.3.4 Execution and Change Detection	56
5.3.5 Implementation	57
5.4 Evaluation	57
5.4.1 Benchmark	57
5.4.2 Protocol	59
5.4.3 Results	59
5.5 Limitations	74
5.6 Threats to validity	75

5.7 Conclusion	75
6 Transversal Contributions	77
7 Conclusion	79
	81

List of Figures

1.1	STAMP H2020 Project Partners	3
1.2	STAMP’s directions integrated in the Software life-cycle.	4
1.3	Dissertation Roadmap	5
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>i.e.</i> the amplified test suite is better than the original one.	17
3.2	Example of what DSpot produces: a diff to improve an existing test case.	18
3.3	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.	18
5.1	Commit 7e79f77 on XWiki-Commons that changes the behavior without a test.	54
5.2	Overview of the approach to detect behavioral changes in commits.	56
5.3	Distribution of diff coverage per project of our benchmark.	62
5.4	Diff of commit 3FADFDD from commons-lang.	63
5.5	Test generated by DCI that detects the behavioral change of 3FADFDD from commons-lang.	63
5.6	Test generated by DCI- <i>I-Amplification</i> that detects the behavioral change introduced by commit 81210EB in commons-io.	69
5.7	Developer test for commit 81210EB of commons-io.	69
5.8	Test generated by DCI- <i>I-Amplification</i> that detects the behavioral change of E7D16C2 in commons-lang.	70
5.9	Developer test for E7D16C2 of commons-lang.	70
5.10	Test generated by DCI that detects the behavioral change of commit 44CAD04 in Gson.	71
5.11	Provided test by the developer for 44CAD04 of Gson.	71
5.12	Test generated by DCI- <i>I-Amplification</i> that detects the behavioral change of 3676B13 of Jsoup.	72
5.13	Provided test by the developer for 3676B13 of Jsoup.	72
5.14	Test generated by DCI- <i>I-Amplification</i> that detects the behavioral change of 774AE7A of Mustache.java.	72

5.15 Developer test for 774AE7A of Mustache.java.	73
5.16 Test generated by DCI-A-Amplification that detects the behavioral change of D3101AE of XWiki.	73
5.17 Developer test for D3101AE of XWiki.	74

List of Tables

4.1	Dataset of 10 active GitHub projects considered on our relevance study (RQ1) and quantitative experiments (RQ2, RQ3).	31
4.2	Overall result of the opened pull request built from result of DSpot.	35
4.3	List of URLs to the pull-requests created in this experiment.	36
4.4	Contributions of <i>A-Amplification</i> and <i>I-Amplification</i> on the amplified test method used to create a pull request.	43
4.5	The effectiveness of test amplification with DSpot on 40 test classes: 24 well-tested (upper part) and 16 average-tested (lower part) real test classes from notable open-source Java projects.	44
5.1	Considered Period for Selecting Commits.	58
5.2	Performance evaluation of DCI on 60 commits from 6 large open-source projects.	60
5.3	Evaluation of the impact of the number of iteration done by <i>DCI-I-Amplification</i> on 60 commits from 6 open-source projects.	65
5.4	Number of amplified test methods obtained by DCI for 10 different seeds. 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 RQ1 : To what extent are <i>DCI-A-Amplification</i> and <i>DCI-I-Amplification</i> able to produce amplified test methods that detect the behavioral changes?. The ten following columns are the result obtained for the 10 different seeds. . .	66
5.5	Standard deviations of the number of amplified tests obtained for each seed.	75

CHAPTER 1

Introduction

Contents

1.1	STAMP-project	2
1.1.1	Research Directions	3
1.1.2	STAMP's Output	4
1.2	Thesis Vision	5
1.3	Thesis Roadmap	5
1.4	Software and Impact On The Community	6
1.5	Publications	7

Nowadays software are omnipresent in people's life: banking, e-commerce, communication, etc. And the more and more, critical aspects such as aeronautics, voting system or even health care relies 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 the Swedish JAS 39 Gripen fighter aircraft.

To avoid such situations, software editors adopted testing philosophies: the tester writes code that 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. 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].

However, testing is tedious and costly for industries: there is no direct return to invest. Thus, developers under pressure or by lack of discipline or time might skip 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 2011,

Pacheco 2005]. The dream was that a command-line would give you a complete test suite, that verifies the whole program. For free, or almost, all the program would be well-tested. However, studies show that developers are not using automatic test generation [Fraser 2015]. The authors investigate the truthfulness of the following hypothesis:

generating high coverage test data, we aid testers in constructing test suites capable of detecting faults.

However, their studies showed that achieving high coverage does not necessarily improve the ability to test software. The difficulties to understand, integrate and maintain automated test suite prevent the adoption by developers. Also, most of the tools relies on weak or partial oracles, *e.g.* absence of runtime errors, making them useless against bugs that do not set the program into a wrong state.

In this thesis, I aim at addressing this issue. More precisely, the ultimate goal of this thesis is to provide an usable tool that assist developers to maintain their test suite, in the context of DevOps and continuous integration. To do so, I use test suite amplification, which is an emerging field, that derived existing test methods to create variants of them according to an engineering goal. It means that test suite amplification exploits the knowledge that developers introduced in the seed test method. This knowledge is a key to obtain usable, valuable and strong test methods. By construction, test suite amplification's output is close to the test methods used seed, since it is obtained by applying insertions, deletions or modifications on the seed test method. Which means that for developers that know very-well their software and their test suite, it is easier to understand amplified test method since they share a commons-part with what they developed.

The remaining of this chapter is as follow:

In [Section 1.1](#) present the context of the thesis, the H2020 European project *STAMP*;

Then, I expose the global vision of this thesis in [Section 1.2](#) and its roadmap in [Section 1.3](#);

Eventually, I list the resulting software of this thesis in [Section 1.4](#) and my publications in [Section 1.5](#).

1.1 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 Software Testing AMPlification. STAMP leverages advanced research in test generation and innovative methods of test amplification to push automation in DevOps one step further.

Test amplification reuses existing test assets and generate more test cases and test con-



Figure 1.1: STAMP H2020 Project Partners

figurations at each modification of the application. 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.

STAMP will raise confidence and foster adoption of DevOps by the European IT industry. 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.

1.1.1 Research Directions

In STAMP, there are 3 research directions: unit testing amplification, configuration testing amplification and online testing amplification. These 3 directions are made to be integrated in different step of the Software life-cycle, see Figure 1.2.

1.1.1.1 Unit Testing Amplification

Unit testing refers to testing small and limited part of the Software. That is to say, it tests a single component, which has, ideally, no interference with others components. Unit testing is a crucial part of testing since it verifies that every single component of Software does what it is expected to do.

It has two majors benefit:

- 1) It is easier to test and verify the behavior of a single component;

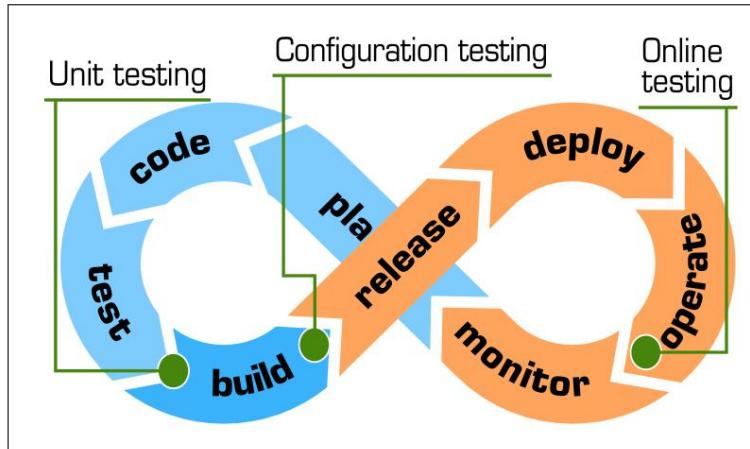


Figure 1.2: STAMP's directions integrated in the Software life-cycle.

- 2) It allows to split the development into small part and view the system as a composition of a lot of small parts.

Unit test amplification aims at generating unit tests that are variants of existing unit tests. These variants would improve the overall quality of unit tests according to a given engineering goal.

During my thesis, I was mainly involved in this research direction.

1.1.1.2 Configuration Testing Amplification

Configuration testing is the activity of assembling different services in a complete system, to deploy and test it, based on a pre-defined model. Configuration testing amplification performs automatic amplification of configurations on the model, in the forms of mutation and crossover. The goal is to create un-tested configuration to stress the Software in newly created environments.

1.1.1.3 Online Amplification

Online test amplification automatically extracts information from logs collected in production in order to generate new tests that can replicate failures, crashes, anomalies and outlier events. This research direction aims at helping developers to debug the Software in case of crash during the production. Debugging is a tedious task, and having a unit test method, automatically generated, would save a lot of time.

1.1.2 STAMP's Output

STAMP's final output would be micro-services that allow the developers to use STAMP's tools on their own project. STAMP envisions a web interface, on which the developers

would be able to install, configure and execute the tools. Also, the developers would be able to manage the output of these executions and link them directly to their project, *e.g.* integrate an amplified unit test into their test suite.

1.2 Thesis Vision

As this thesis is funded by STAMP, its ultimate goal is correlated to the one of STAMP. That is to say, this thesis aims at providing a set of tools to assist developers in the test suite evolution. In particular, these tools apply test amplification to generate test methods. These amplified test methods achieve diverse goals, such as improving the overall quality of the test suite.

The first objective is to improve the test suite “offline”, *i.e.* outside the continuous integration service. This aims at showing that the tool is effectively able to improve the test suite according to an engineering such as a measure of the test suite’s quality.

The second objective is to obtain test methods that are able to characterize a behavioral changes of the program. 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.

1.3 Thesis Roadmap

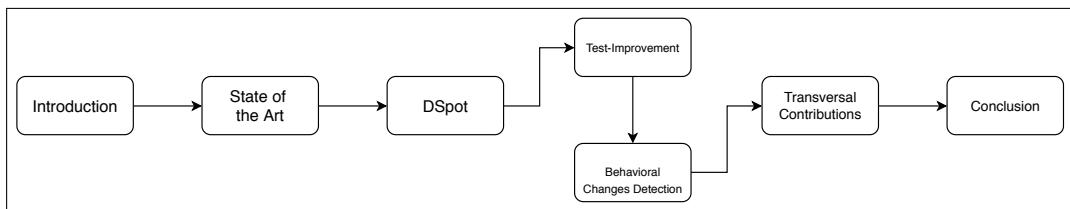


Figure 1.3: Dissertation Roadmap

This thesis is divided into 6 chapters.

First, [Chapter 2](#) exposes the state of the art of test amplification.

Then, [Chapter 3](#) gives the detail of the major technical contributions of this thesis: DSpot.

[Chapter 4](#) and [Chapter 5](#) relate two evaluations of DSpot on open-source projects from GitHub. Each of these evaluations has their own characteristics, novelties and motivations.

During my thesis, I developed a lot of skills that allowed me to participate to transversal contributions. These contributions are related in [Chapter 6](#).

Eventually, I conclude and give the short- and long-term perspectives in [Chapter 7](#).

[Figure 1.3](#) summarizes the roadmap.

1.4 Software and Impact On The Community

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

DSpot¹ 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.

DSpot-diff-test-selection² is a maven plugin, based on OpenClover, that produces the list of test classes and their test methods that execute a provided diff.

DSpot-prettifier³ aims a make better good looking amplified test methods obtained with DSpot. It relies on several operations such as minimization, renaming local variables and test methods (based on code2vec)

Test-runner⁴ is a library that allows developer to execute test in a new and clean JVM.

AssertFixer⁵ aims to fix the assertion in a test suite, using the program as specifications.

To maximize the artefacts' impact on the community, all of them are open-source and available on GitHub.

¹<https://github.com/STAMP-project/dspot.git>

²<https://github.com/STAMP-project/dspot/tree/master/dspot-diff-test-selection>

³<https://github.com/STAMP-project/dspot/tree/master/dspot-prettifier>

⁴<https://github.com/STAMP-project/test-runner>

⁵<https://github.com/STAMP-project/AssertFixer>

1.5 Publications

- [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 27 and 57.)
- [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 51.)
- [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 57 and 77.)
- [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 page 77.)
- [Danglot 2017] Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Martin Monperrus and Benoit Baudry. *The Emerging Field of Test Amplification: A Survey*. CoRR, vol. abs/1705.10692, 2017. (Cited on page 9.)
- [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 77.)

CHAPTER 2

State of the Art

In this chapter, I exposed a systematic review of the literature on the field of test suite amplification. I surveyed works that exploit this knowledge to enhance manually written tests with respect to an engineering goal (*e.g.* improve coverage or refine fault localization). This chapter provides 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 is a to be published article[[Danglot 2017](#)]. The remainder of this chapter is as follows: This chapter is structured according to the 4 main categories, each of them being presented in a dedicated section. Section ?? presents techniques that synthesize new tests from manually-written tests. Section ?? focuses on the works that synthesize new tests dedicated to a specific change in the application code (in particular a specific commit). Section ?? discusses the less-researched, yet powerful idea of modifying the execution of manually-written tests. Section ?? is about the modification of existing tests to improve a specific property.

Contents

2.1	Introduction	9
2.2	Method	10
2.2.1	Definition	11
2.2.2	Methodology	11
2.2.3	Novelty	13

2.1 Introduction

Software testing is the art of evaluating an attribute or capability of a program to determine that it meets its required results [Hetzl 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 surveys the research literature so that existing research efforts are characterized, can be compared and new research opportunities can be identified. Furthermore, the conjecture is that with good foundations and maturation, test amplification has the potential to bring software testing to the next level in terms of efficiency and efficacy among practitioners by introducing new automatic processes that improve the manually written tests.

The reviewing methodology is based on backward- and forward- snowballing on the citation graph [Jalali 2012]. To the best of my knowledge, this review is the first that draws a comprehensive picture of the different engineering techniques and goals proposed in the literature for test amplification.

2.2 Method

In this section, we present the methodology of our systematic literature review.

2.2.1 Definition

In this review, we use the following definition of test amplification:

Definition: Test amplification consists of exploiting the knowledge of a large number of test cases, 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 cases automatically generated from the existing manual test cases 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.**

2.2.2 Methodology

Literature studies typically rigorously follow a methodology to ensure both completeness and replication. We refer to Cooper’s book 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*, we found that 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, we perform our study per Wohlin’s guidelines [Wohlin 2014, Jalali 2012].

First, we 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. We have reviewed them one by one to see whether they fit in our scope according to the definition of subsection 2.2.1. This has resulted in four 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 of our paper.

We now briefly describe these four seed papers. More details are given in the following sections.

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

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

- [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.

From the seed papers, we have performed a backward snowballing search step [Jalali 2012], i.e., we have looked at all their references, going backward in the citation graph. Two of the authors have reviewed the papers, independently. Then, these 2 authors cross-checked the outcome of their literature review, and kept each paper for which they both agreed that it fits the definition of test amplification (cf. subsection 2.2.1). Then, we have performed a forward literature search step, 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 we call an “iteration”. With each iteration, we select a set of papers for our study that we obtain through the snowballing action. We iterate until this set of selected paper is empty, i.e., when no paper can be kept, we stop the snowballing process in both ways: backward and forward.

Once we had selected the papers for our study, we distinguished 4 key approaches to amplification, which we use to classify the literature : amplification by Adding New Tests as Variants of Existing Ones (??); Amplification by Modifying Test Execution (??); Amplification by Synthesizing New Tests with Respect to Changes (??); Amplification by Modifying Existing Test Code (??). The missing terminological consensus mentioned previously prevented the design of a classification according to Petersen’s guidelines [Petersen 2008]. Incrementally, we have refined the four categories by analyzing the techniques and goals in each paper. Our methodology is as follows: we assign a work to a category if the key technique of the paper corresponds to it, per a consensus between the authors. If no category captures the gist of the paper, we have created a new category. If two categories are found to be closely related, we merge both categories to create a new one. The incremental refinement of these findings led to the definition of four categories to organize this literature study.

2.2.3 Novelty

There are a number of notable surveys in software testing [Edvardsson 1999, McMinn 2004, Anand 2013]. However none of them is dedicated to test amplification.

For instance, we refer to Edvardsson’s et al’s [Edvardsson 1999] and McMinn et al’s [McMinn 2004] articles for a survey on test generation. Yoo and Harman have structured the work on test minimization, selection and prioritization [Yoo 2012]. In the prolific literature on symbolic execution for testing, we refer the reader to the survey of Păsăreanu and Visser [Păsăreanu 2009].

In general, test optimization, test selection, test prioritization, test minimization, test reduction is out of the scope of this paper.

Similarly, the work on test refactoring is related, but not in scope. In particular, the work from Van Deursen et al. [van Deursen 2001, Moonen 2008] and Mesaros [Meszaros 2006] focuses on improving the structural and diagnosability qualities of software tests, and is a mainly manual activity. In contrast, test amplification is meant to be fully automated, as other technical amplification such as sound amplification. Its goal is also different, in that its aim is to test more effectively with regard to a given target criterion.

Harrold et al. [Harrold 2008] discusses the problem of “retesting software”, where there is a section related to amplification. However, it is only a light account on the topic which is now outdated. To our knowledge, this survey is the first survey ever dedicated to test amplification.

Yusifoğlu et al. [Yusifoğlu 2015] discuss the new trends in software test-code engineering, and discuss the implications for researchers and practitioners in this area. To do this, they use a systematic mapping to identify areas that require more attention. Their work covers a larger scope than our work, since they study all software test-code engineering research, methods and empirical study, while we focus specifically on test amplification, with more depth.

CHAPTER 3

DSpot: A Test Amplification Tool

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

1. it modifies the test inputs in order to trigger new behavior.
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.

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](#)

Contents

3.1 Definitions	16
3.2 Overview	17
3.2.1 Principle	17
3.2.2 Input & Output	17
3.2.3 Workflow	18
3.2.4 Test method example	19
3.3 Algorithm	20
3.3.1 Input Space Exploration Algorithm	20
3.3.2 Assertion Improvement Algorithm	21
3.3.3 Pseudo-algorithm	22
3.3.4 Flaky tests elimination	23

3.4 Implementation	24
3.4.1 Ecosystem	24
3.5 Conclusion	24

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

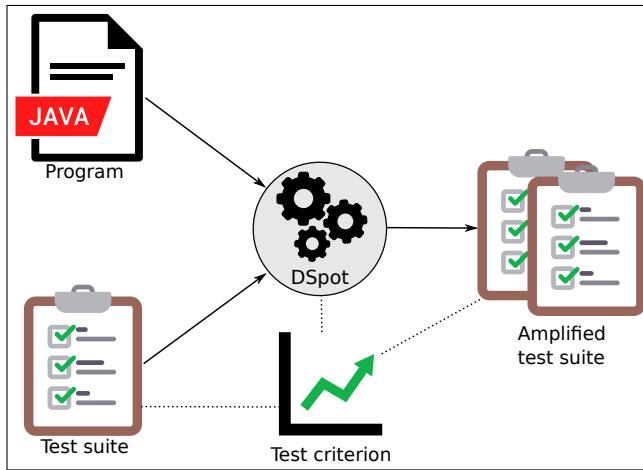


Figure 3.1: DSpot’s principle: DSpot takes as input a program, an existing test suite, and a test-criterion. DSpot outputs a set of amplified test methods. When added to the existing test suite, these amplified test methods increase the test-criterion, *i.e.* the amplified test suite is better than the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

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.

```
writeListTo(out, foos, SerializableObjects.foo.cachedSchema());
final int bytesWritten = writeListTo(out, foos, SerializableObjects.foo.cachedSchema());
assertEquals(0, bytesWritten);
```

Figure 3.2: Example of what DSpot produces: a diff to improve an existing test case.

[Figure 3.2](#) 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

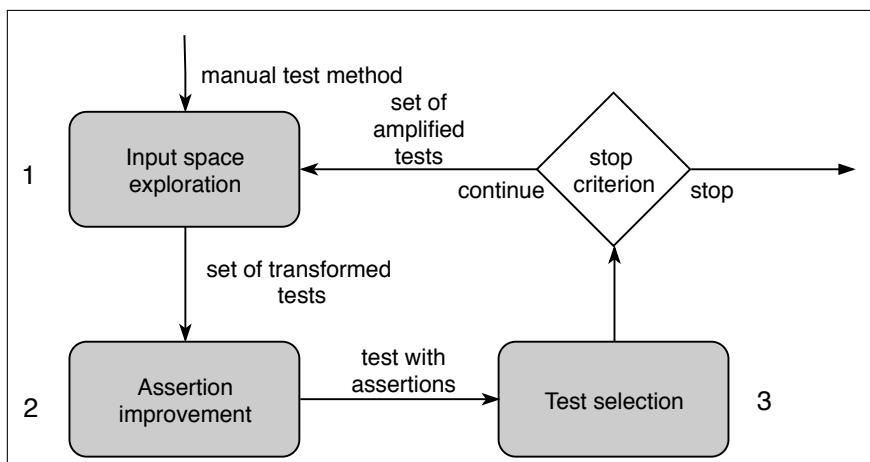


Figure 3.3: DSpot's workflow in three main steps: 1) the modification of test code's inputs, called “input space exploration”; 2) the addition of new assertions called “assertion improvement”; 3) the amplified test methods selection according to a test-criterion.

```

1 testIterationOrder() {
2     // contract: the iteration order is the same as the insertion
3     // order
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 }
```

Listing 3.1: An example of an object-oriented test case (inspired from Apache Commons Collections)

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 exploration”; 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 2006a], 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. 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.

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 3.1](#) 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.

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).

For numeric values, there are five operators: $+1$, -1 , replacement by hard-coded values: max value, min value, 0, and replacement by an existing literal of the same type, if such literal exists.

For Strings, there are seven operators: add a random char, remove a random char, replace a random char by a random char, replace the string by a fully random string of the same size, replace the string by an empty string, replace the string by system line separator and replace the string by the system path separator.

For booleans, there is only one operator: negate the value;

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 a new object of the required type using the default constructor if it exists. 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 3.1, it may generate a new method call on *tl*. In Listing 3.2, the added method call is “removeAll”.

```

1 testIterationOrder() {
2     TreeList tl=new TreeList();
3     tl.add(1);
4     tl.add(2);
5     tl.removeAll(); // method call added
6
7     // removed assertions
8 }
```

Listing 3.2: An example of an *I-Amplification*: the amplification added a method call to `removeAll()` on `tl`.

Since DSpot changes the state of the program, existing assertions may fail. That is why it removes also all existing 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 explores 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

To improve existing tests, DSpot adds new assertions as follows.

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 results 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.

For example, let consider *A-Amplification* on the test method of the example above.

First, in Listing 3.3 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 3.4, DSpot has generated two new assertions.

```

1 testIterationOrder() {
2   TreeList tl=new TreeList();
3   tl.add(1);
4   tl.add(2); aampl
5   tl.removeAll();
6
7   // logging current behavior
8   Observations.observe(tl.size());
9   Observations.observe(tl.isEmpty());
10 }

```

Listing 3.3: In *A-Amplification*, the second step is to instrument and run the test to collect runtime values.

```

1 testIterationOrder() {
2   TreeList tl=new TreeList();
3   tl.add(1);
4   tl.add(2);
5   tl.removeAll();
6
7   // generated assertions
8   assertEquals(0, tl.size()); // generated assertions
9   assertTrue(tl.isEmpty()); // generated assertions
10 }

```

Listing 3.4: In *A-Amplification*, the last step is to generate the assertions based on the collected values.

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 $n > 0$

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:  $ATS \leftarrow \emptyset$ 
2: for  $t$  in  $TS$  do
3:    $U \leftarrow generateAssertions(t)$ 
4:    $ATS \leftarrow \{x \in U | x \text{ improves } TC\}$ 
5:    $TMP \leftarrow ATS$ 
6:   for  $i = 0$  to  $n$  do
7:      $V \leftarrow []$ 
8:     for  $amp$  in  $amps$  do
9:        $V \leftarrow V \cup amp.apply(TMP)$ 
10:    end for
11:     $V \leftarrow generateAssertions(V)$ 
12:     $ATS \leftarrow ATS \cup \{x \in V | x \text{ improves } TC\}$ 
13:     $TMP \leftarrow V$ 
14:  end for
15: end for return  $ATS$ 

```

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.

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 woper showed their interest in amplified test methods.here 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 runs 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 Github¹. 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 presents technical details about DSpot. DSpot is a test amplification tool that improves the test suite. DSpot works in three main steps:

1. it modifies the test inputs in order to trigger new behavior.
2. it generates assertions to verify the new behavior of the program.

¹<https://github.com/STAMP-project/dspot>

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.

In the two following chapters, I evaluate the performance of DSpot to improve existing test suite in two scenarios:

A first scenario where DSpot improve existing test suite 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 in order 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

In this chapter, I detail a first evaluation of DSpot. This evaluation is based on the mutation score as test-criterion. Mutation score measures the test suite's ability to detect artificial behavioral changes. I confronted DSpot's output to real projects from GitHub. To do so, I propose to developers to integrate directly the amplified test methods into their test suite. Developers showed their interest in amplified test methods by accepting permanently 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 projects;
- 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^a

Note that this chapter has been published [Danglot 2019b].

The remainder of this chapter is as follows: [Section 4.1](#) introduces this chapter; [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;

^a<https://github.com/STAMP-project/dspot-experiments/>

Contents

4.1 Introduction	28
4.1.1 Test-criterion: mutation score	28

4.1.2	Mutation score versus coverage	29
4.2	Experimental Protocol	29
4.2.1	Dataset	30
4.2.2	Test Case Selection Process	30
4.2.3	Metrics	32
4.2.4	Methodology	32
4.3	Experimental Results	33
4.3.1	Answer to RQ1	33
4.3.2	Answer to RQ2	43
4.3.3	Answer to RQ3	46
4.3.4	Answer to RQ4	48
4.4	Threats to Validity	49
4.5	Conclusion	49

4.1 Introduction

4.1.1 Test-criterion: mutation score

This chapter relates the first study of the DSpot’s effectiveness. To do this evaluation, I choose mutation score as test-criterion to be improved by DSpot. Mutation score measures the test suite’s ability to detect artificial behavioral changes. Briefly, mutation score 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 one 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 Pitest¹ because:

- 1) it targets Java programs;
- 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/>.

4.1.2 Mutation score versus coverage

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 2006a]. However:

- 1) the two topics have never been studied in conjunction
- 2) they have never been studied on large modern Java programs

¹latest version released at the time of the experimentation: 1.2.0.<https://github.com/hcoles/pitest/releases/tag/1.2.0>

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:

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?

4.2.1 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](#).

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 [subsection 4.2.1](#). 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.

4.2.2 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

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 OdbDecoderTest ^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 with Twilio 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

Table 4.1: Dataset of 10 active GitHub projects considered on our relevance study (RQ1) and quantitative experiments (RQ2, RQ3).

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 subsection 4.1.1);
- 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 4.2.3), 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 subsection 4.2.1, 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.

4.2.3 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.

4.2.4 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 subsection 4.1.1) 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*.²

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?

²Note 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*).

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 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 have been created, as shown in [Table 4.2](#). 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, *i.e.* nor merged nor closed. Note that these numbers might change over time if pull-requests are merged or closed.

[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

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 the following one liner pull request ³:

³<https://github.com/square/javapoet/pull/544>

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

```

181      assertEquals(a.hashCode(), b.hashCode());
182 +     assertFalse(a.equals(null));

```

The title of the pull request 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 have proposed to the developers the following pull request 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.*”⁴:

```
diff --git a/src/main/java/org/apache/struts2/meta/MetaClass.java b/src/main/java/org/apache/struts2/meta/MetaClass.java
--- a/src/main/java/org/apache/struts2/meta/MetaClass.java
+++ b/src/main/java/org/apache/struts2/meta/MetaClass.java
@@ -68,6 +68,10 @@ public void buildProperty(String name, StringBuilder sb) {
     + assertEquals("richType.richProperty", meta.findProperty("richType.richProperty", false));
     +
@@ -70,6 +70,8 @@ public void buildProperty(String name, StringBuilder sb) {
     + assertFalse(meta.hasGetter("[0]"));
 }
```

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 `decodeCodes()` 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*”.⁵

```
diff --git a/test/java/org/traccar/test/ObdDecoderTest.java b/test/java/org/traccar/test/ObdDecoderTest.java
--- a/test/java/org/traccar/test/ObdDecoderTest.java
+++ b/test/java/org/traccar/test/ObdDecoderTest.java
@@ -17,6 +17,7 @@ public class ObdDecoderTest {
@@ -18,6 +18,7 @@ public class ObdDecoderTest {
@@ -19,6 +19,10 @@ public class ObdDecoderTest {
     + @Test
     + public void testDecodeCodes() throws Exception {
@@ -20,6 +20,10 @@ public class ObdDecoderTest {
     +     Assert.assertEquals("P0D14", ObdDecoder.decodeCodes("0D14").getValue());
     +     Assert.assertEquals("dtcs", ObdDecoder.decodeCodes("0D14").getKey());
     + }
@@ -21,6 +21,7 @@ public class ObdDecoderTest {
     + }
@@ -22,6 +22,7 @@ public class ObdDecoderTest {
     + }
@@ -23,6 +23,7 @@ public class ObdDecoderTest {
     + }
@@ -24,6 +24,7 @@ public class ObdDecoderTest {
     + }
@@ -25,6 +25,7 @@ public class ObdDecoderTest {
 }
```

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

⁴<https://github.com/mybatis/mybatis-3/pull/912/files>

⁵<https://github.com/tananaev/traccar/pull/2897>

MurmurHash from lines 158 to 216. This method computes a hash for a given array of byte. The PR 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.*”.⁶:

```

47 -     long hashOfString = MurmurHash.hash64(input);
47 +     long hashOfString = -8896273065425798843L;
48     assertEquals("MurmurHash.hash64(byte[]) did not match MurmurHash.hash64(String)",
49             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 a 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.⁷:

```

54     }
55 +
56 +    // test the returned valued of hash functions against the reference implementation: https://github
57 +
58 +    @Test
59 +    public void testHash64() throws Exception {
60 +        final long actualHash = MurmurHash.hash64("hashthis");
61 +        final long expectedHash = -8896273065425798843L;
62 +
63 +        assertEquals("MurmurHash.hash64(String) returns wrong hash value", expectedHash, actualHash);
64 +    }
65 +
66 +    @Test
67 +    public void testHash() throws Exception {
68 +        final long actualHash = MurmurHash.hash("hashthis");
69 +        final long expectedHash = -1974946086L;
70 +
71 +        assertEquals("MurmurHash.hash(String) returns wrong hash value", expectedHash, actualHash);
72 +    }
73 } @@

```

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

⁶<https://github.com/addthis/stream-lib/pull/127/files>

⁷<https://github.com/addthis/stream-lib/pull/128/files>

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 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.*”.⁸

```

68      }
69      +
70      + @Test
71      + public void testImproperlyClosedVariable() throws IOException {
72      +     try {
73      +         new DefaultMustacheFactory().compile(new StringReader("{{#containers}} {{/containers}}"), "example");
74      +         fail("Should have throw MustacheException");
75      +     } catch (MustacheException actual) {
76      +         assertEquals("Improperly closed variable in example:1 @[example:1]", actual.getMessage());
77      +     }
78      + }
79      +
80  }
```

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.⁹.

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*”¹⁰:

⁸<https://github.com/spullara/mustache.java/pull/186/files>

⁹the diff is same:<https://github.com/spullara/mustache.java/commit/9efea19d595f893527ff218683e70db2ae4d8fb2d>

¹⁰<https://github.com/twilio/twilio-java/pull/334/files>

```

168
169 +     @Test
170 +     public void testEquals() {
171 +         Request request = new Request(HttpMethod.DELETE, "/uri");
172 +         request.setAuth("username", "password");
173 +         assertFalse(request.equals(new Object()));
174 +         assertFalse(request.equals(null));
175 +     }
176 +
177 }
```

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 was entitled “*add test case for hashCode in attribute*” with the following short description “*Hello, I propose this change to specify the hashCode of the object org.jsoup.nodes.Attribute.*”¹¹:

```

19     }
20 +
21 +     @Test
22 +     public void testHashCode() {
23 +         String s = new String(Character.toChars(135361));
24 +         Attribute attr = new Attribute(s, ((A + s) + B));
25 +         assertEquals(111849895, attr.hashCode());
26 +     }
27 }
```

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

¹¹<https://github.com/jhy/jsoup/pull/840>

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 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.*”¹², shown earlier in Figure 3.2

```

146      ByteArrayOutputStream out = new ByteArrayOutputStream();
147 -     writeListTo(out, foos, SerializableObjects.foo.cachedSchema());
148 +     final int bytesWritten = writeListTo(out, foos, SerializableObjects.foo.cachedSchema());
149 +     assertEquals(0, bytesWritten);
150
151     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*”.¹³:

¹²<https://github.com/protostuff/protostuff/pull/212/files>

¹³<https://github.com/qos-ch/logback/pull/365/files>

```

192    }
193 +
194 + @Test
195 + public void testNotEqualsNull() {
196 +     FileNamePattern pp = new FileNamePattern("t", context);
197 +     assertFalse(pp.equals(null));
198 +
199 +
200 }
```

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.

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

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

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 4.2.2. The seventh,

Table 4.5: The effectiveness of test amplification with DSpot on 40 test classes: 24 well-tested (upper part) and 16 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	1250%	19	8 599715	19% ↗	599	0.0% →	11.11		
2	NameAllocatorTest	1187%	0	0 79 79	0.0% →	79	0.0% →	4.76		
3	MetaClassTest	758%	108	10 455534	17% ↗	455	0.0% →	235.71		
4	ParameterExpressionTest	1491%	2	2 162164	1% ↗	162	0.0% →	25.93		
5	ObdDecoderTest	180%	9	2 51 54	5% ↗	51	0.0% →	2.20		
6	MiscFormatterTest	172%	5	5 42 47	11% ↗	42	0.0% →	1.21		
7	TestLookup3Hash	295%	0	0 4644464	0.0% →	464	0.0% →	6.76		
8	TestDoublyLinkedList	792%	1	1 1041050.97%	↗	104	0.0% →	3.03		
9	ArraysIndexesTest	153%	15	4 576647	12% ↗	586	1% ↗	10.58		
10	ClasspathResolverTest	1067%	0	0 50 50	0.0% →	50	0.0% →	4.18		
11	RequestTest	1781%	4	3 141156	10% ↗	141	0.0% →	60.55		
12	PrefixedCollapsibleMapTest	496%	0	0 54 54	0.0% →	54	0.0% →	13.28		
13	TokenQueueTest	669%	18	6 152165	8% ↗	152	0.0% →	15.61		
14	CharacterReaderTest	1979%	71	9 309336	8% ↗	309	0.0% →	57.06		
15	TailDelimiterTest	1071%	1	1 3813840.79%	↗	381	0.0% →	12.90		
16	LinkBufferTest	348%	12	7 66 90	36% ↗	66	0.0% →	3.24		
17	FileNamePatternTest	1258%	27	9 573686	19% ↗	573	0.0% →	25.08		
18	SyslogAppenderBaseTest	195%	1	1 143148	3% ↗	143	0.0% →	7.88		
19	RequestBuilderAndroidTest	299%	0	0 513513	0.0% →	513	0.0% →	0.04		
20	CallAdapterTest	494%	0	0 55 55	0.0% →	55	0.0% →	7.30		
Low mutation score										
21	FieldSpecTest	231%	12	4 223316	41% ↗	223	0.0% →	4.44		
22	ParameterSpecTest	232%	11	5 214293	36% ↗	214	0.0% →	3.66		
23	WrongNamespacesTest	2 8%	6	1 78249	219% ↗	249	219% ↗	29.70		
24	WrongMapperTest	1 8%	3	1 97325	235% ↗	325	235% ↗	7.13		
25	ProgressProtocolDecoderTest	116%	2	1 18 27	50% ↗	23	27% ↗	1.30		
26	IgnitionEventHandlerTest	122%	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	217%	40	2 52275	428% ↗	174	234% ↗	2.18		
29	ConcurrencyTest	228%	2	0 210342	62% ↗	210	0.0% →	315.56		
30	AbstractClassTest	234%	28	4 383475	24% ↗	405	5% ↗	12.67		
31	AllTimeTest	342%	0	0 163163	0.0% →	163	0.0% →	0.02		
32	DailyTest	342%	0	0 163163	0.0% →	163	0.0% →	0.02		
33	AttributeTest	236%	33	11 178225	26% ↗	180	1% ↗	10.76		
34	AttributesTest	552%	9	6 316322	1% ↗	316	0.0% →	6.21		
35	CodedDataInputTest	1 1%	0	0 5 5	0.0% →	5	0.0% →	3.58		
36	CodedInputTest	127%	29	28 108166	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	110%	0	0 6 6	0.0% →	6	0.0% →	0.89		
39	ExecutorCallAdapterFactoryTest	762%	0	0 119119	0.0% →	119	0.0% →	0.09		
40	CallTest	3569%	3	1 6426440.32%	↗	642	0.0% →	52.84		

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, 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 classes 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 2006b], 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

To conclude, this chapter relates a first experimentation on DSpot:

This evaluation is two folds:

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

2) 40 test classes has 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 suite.

In this chapter, mutation score has been used to amplified test methods. Mutants are small and artificial behavioral changes in the code. DSpot is able to improve the existing test methods' detection ability of small and artificial changes.

In the next chapter, I investigate the capacity of DSpot to improve existing test methods in order to detect real behavioral changes. 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 places it in the continuous integration and regression testing scenarios.

CHAPTER 5

Test Amplification For Commit Behavioral Changes Detection

In this chapter, I detail an extension of DSpot, called DCI (DSpot-CI), and its evaluation. When developers use a version control management such as git, they make changes in the software in the form of commits. DCI consists of obtaining amplified test methods that detect the behavioral changes introduced by commits. This evaluation has been done on 60 commits from 6 open-source projects on GitHub. The result is that DCI has been able to obtain amplified test methods detecting **XX** behavioral changes.

To sum up, the contributions of this chapter are:

- DCI (**Dspot-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^a.
- 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 5.2](#) motivates this chapter and gives the background. [Section 5.3](#) exposes the technical extension of DSpot: an approach for commit-based test selection. [Section 5.4](#) introduces our benchmark of commits, the evaluation protocol and the results of our experiments on 50 real commits. [Section 5.6](#) relates the threats validity and actions that have been taken to overcome them. and [Section 5.7](#) concludes this chapter;

^a<https://github.com/STAMP-project/dspot-experiments.git>

Contents

5.1	Introduction	52
5.1.1	Collaborative software development	52
5.1.2	Goal	53
5.2	Motivation & Background	53
5.2.1	Motivating Example	53
5.2.2	Practicability	54
5.2.3	Behavioral Change	55
5.2.4	Behavioral Change Detection	55
5.3	Behavioral Change Detection Approach	55
5.3.1	Overview of DCI	55
5.3.2	Test Selection and Diff Coverage	56
5.3.3	Test Amplification	56
5.3.4	Execution and Change Detection	56
5.3.5	Implementation	57
5.4	Evaluation	57
5.4.1	Benchmark	57
5.4.2	Protocol	59
5.4.3	Results	59
5.5	Limitations	74
5.6	Threats to validity	75
5.7	Conclusion	75

5.1 Introduction

5.1.1 Collaborative software development

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.

5.1.2 Goal

The goal is to automatically generate test methods for each commit that is submitted to the CI. In particular, a test method t_{gen} that specifies the behavioral change of each commit. A generated test method t_{gen} is considered to be relevant if it satisfies the following property: t_{gen} *passes* on the pre-commit version and *fails* on the post-commit version. To do so, I developed a new approach, called DCI (**D**etecting behavioral changes in **CI**), and propose it be used during continuous integration., that works in two steps: First, it analyzes the test methods of the pre-commit version and select 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.

5.2 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.

5.2.1 Motivating Example

On August 10, a developer pushed a commit to the master branch of the XWiki-commons project. The change¹, 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.

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

¹<https://github.com/xwiki/xwiki-commons/commit/7e79f77>

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

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

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 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.

5.2.2 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 subsection 5.1.2, 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 2009].

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.

5.2.3 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.

5.2.4 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 behavioral changes based on test amplification.

5.3 Behavioral Change Detection Approach

5.3.1 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.

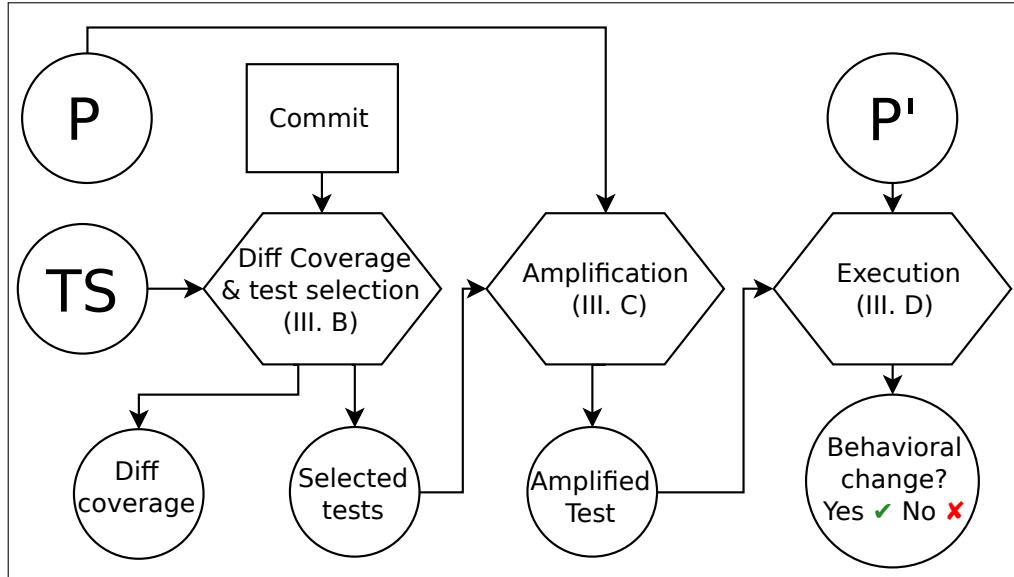


Figure 5.2: Overview of the approach to detect behavioral changes in commits.

5.3.2 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.

5.3.3 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*.

5.3.4 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.

5.3.5 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.²

5.4 Evaluation

The evaluation of DCI relies on 4 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?

5.4.1 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 2018] and [Danglot 2019b], which is composed of mature Java projects from GitHub.

²<https://github.com/STAMP-project/dspot/tree/master/dspot-diff-test-selection>

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: Considered Period for Selecting Commits.

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

Final benchmark **Table 5.1** shows the main statistics on the benchmark dataset. The first column is the name of the considered project; The second column is the date at which the analysis has been done; The third column is the date of the oldest commit for the project; The fourth, fifth, sixth and seventh are respectively the total number of commit analyzed, the total number of commits discarded, the number of commits that match all

criteria but the third (there is no test in the pre-commit that execute the change). Note that this benchmark is available on GitHub at <https://github.com/STAMP-project/dspot-experiments>.

5.4.2 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.

The null hypothesis is the following:

The population median of all of the groups are equals.

The alternative hypothesis is: at least one population median of one group is different from the population median of at least one other group.

The confidence level is set at 95%, or $\alpha = 0.05$.

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.

5.4.3 Results

The overall results are reported in [Table 5.2](#). The first column is the shortened commit id; the second column is the commit date; the third column column is the total number of test methods executed when building that version of the project; the fourth and fifth columns are 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 sixth and seventh columns are respectively the diff coverage and the number of tests DCI selected; the eighth 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 ninth column displays the time spent on the amplification phase; The tenth and eleventh columns are respectively a ✓ with the number of amplified tests for DCI-*I-Amplification* or a - if DCI did not succeed in generating a test that detects a change; for 3 iterations.

5.4.3.1 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.

5.4.3.2 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 **RQ₁**. 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 (4th row). In the other hands, only DCI-*I-Amplification* has been able to obtain amplified test methods for COMMONS-IO#81210EB (8th 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.

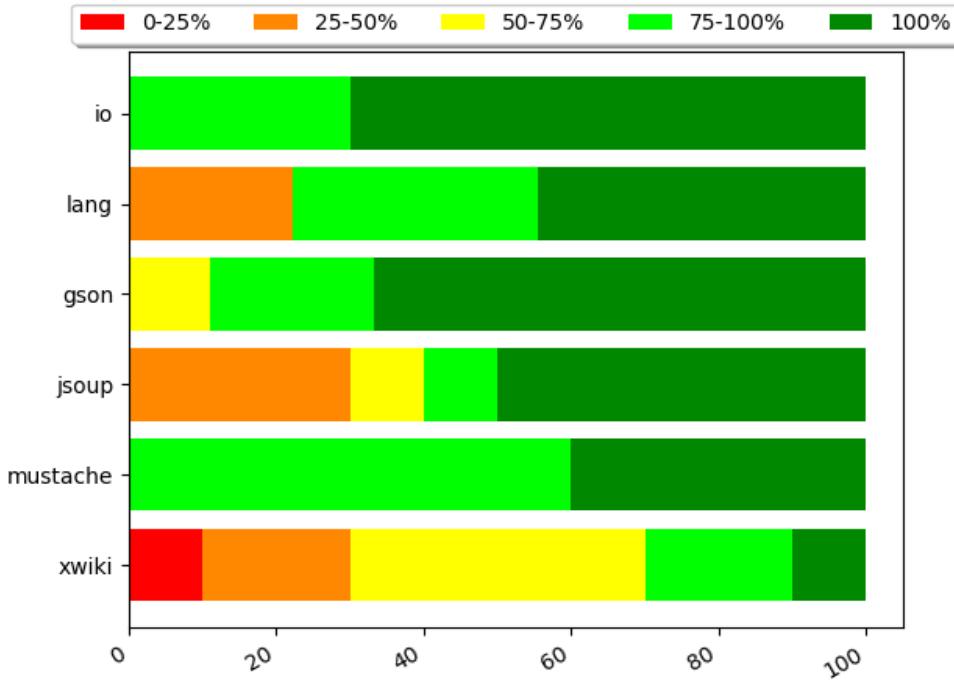


Figure 5.3: Distribution of diff coverage per project of our benchmark.

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 3FADFDD³ from commons-lang has been selected because it is succinct enough to be discussed. The diff is shown in Figure 5.4.

```

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

```

Figure 5.4: Diff of commit 3FADFDD from commons-lang.

The developer added a method call to a method that escapes special 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`. The test is selected for amplification because it triggers the execution of the changed line.

```

StringBuilder o_testNestingPerson_add33752_20 =
    new StringBuilder(nestP).append("pid", nestP.pid).append("per/on",
Assert.assertEquals("{\"pid\":\"#1@Jane\", \"per/on\":{\"name\":\"Jane Doe\"");

```

Figure 5.5: Test generated by DCI that detects the behavioral change of 3FADFDD from commons-lang.

The resulting amplified test method is shown in Figure 5.5. From this test method, DCI generates an amplified test method shown in Figure 5.5. 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 tests the behavioral change: in the pre-commit version, the expected value is: " { ... per/on": {"name": "Jane Doe" ... } " while in the post-commit version it is " { ... per\!/on": {"name": "Jane Doe" ... } " (Line 3).

Answer to 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

³<https://github.com/apache/commons-lang/commit/3fadfdd>

time.

5.4.3.3 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.

Overall, DCI-*I-Amplification* generates amplified tests that detect 21, 23, and 24 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 3 more behavioral changes than when it is used with *iteration* = 1 and 1 more than when it is used with *iteration* = 2. It represents an increase of 14% and 4% for respectively *iteration* = 1 and *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.

Impact of the randomness The number of amplified test methods obtained by the different seeds are reported in Table 5.4. The result of the Kruskal-Wallis test is: *p-value* = 0.96. *p-value* > α which means that the null hypothesis holds: The population median of all of the groups are equal. On other words, the choice of the seeds has not a significant impact of the overall result of DCI.

Answer to RQ2: DCI-*I-Amplification* detects 21, 23, and 24 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.

5.4.3.4 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 ??, 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#C6B8A38⁴: 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#F56931C⁵: the test selection returns 39 test methods from 5 test classes: `FastDateFormat_ParserTest`, `FastDateParserTest`, `DateUtilTest`, `FastDateParser_TimeZoneStrategyTest` and `FastDateParser_MoreOrLessTest`. This commit modifies the behavior of two methods: `simpleQuote` 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#9E6F2BA⁶: 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#E9FEEC9⁷, the test selection returns the 4 test methods defined in the `XmlTreeBuilderTest` class `caseSensitiveDeclaration`, `handlesXmlDeclarationAsDeclaration`, `testDetectCharsetEncodingDeclaration` and `testParseDeclarationAttributes`. The commit modifies the behavior of the class `XmlTreeBuilder`. Here, the test selection is relevant. Actually, the ground-truth

⁴<https://github.com/apache/commons-io/commit/c6b8a38>

⁵<https://github.com/apache/commons-lang/commit/f56931c>

⁶<https://github.com/google/gson/commit/9e6f2ba>

⁷<https://github.com/jhy/jsoup/commit/e9feec9>

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#88718BC⁸, 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#848C984⁹ the test selection returns a single test method `createReference` 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.

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.

5.4.3.5 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#81210EB¹⁰: This commit modifies the behavior of the `read()` method in `BoundedReader`. Figure 5.6 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*

⁸<https://github.com/spullara/mustache.java/commit/88718bc>

⁹<https://github.com/xwiki/xwiki-commons/commit/848c984>

¹⁰<https://github.com/apache/commons-io/commit/81210eb>

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.

```
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)));
```

Figure 5.6: Test generated by DCI-*I-Amplification* that detects the behavioral change introduced by commit 81210EB in commons-io.

Now, look at the human test contained in the commit, shown in Figure 5.7. 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.

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

Figure 5.7: Developer test for commit 81210EB of commons-io.

COMMONS-LANG#E7D16C2¹¹: this commit escapes special characters before adding them to a `StringBuffer`. Figure 5.8 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 \.

```
new StringBuilder(this.base).append("a", '\n').append("b", 'B');
Assert.assertEquals("{\"a\":\"\n\", \"b\":\"B\",",
    ((StringBuffer) o_testChar_add45986_10.getStringBuffer()).toString())
```

Figure 5.8: Test generated by DCI-*I-Amplification* that detects the behavioral change of E7D16C2 in commons-lang.

Let's have a look to the human test method shown in Figure 5.9. 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.

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

Figure 5.9: Developer test for E7D16C2 of commons-lang.

GSON#44CAD04¹²: This commit allows Gson to deserialize a number represented as a string. Figure 5.10 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). ¹³.

¹¹<https://github.com/apache/commons-lang/commit/e7d16c2>

¹²<https://github.com/google/gson/commit/44cad04>

¹³Interestingly, the number is parsed lazily, only when needed. Consequently, the exception is thrown when invoking the `longValue()` method and not when invoking `parse()`

```

    json = "dhs";
    Assert.assertEquals("dhs", json);
    actual = this.gson.fromJson(json, Number.class);
    actual.longValue();
    org.junit.Assert.fail("testNumberDeserialization");
} catch (JsonSyntaxException expected) {
}

```

Figure 5.10: Test generated by DCI that detects the behavioral change of commit 44CAD04 in Gson.

The amplified test methods is now compared against the developer-written ground-truth method, shown in Figure 5.11. 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.

```

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

```

Figure 5.11: Provided test by the developer for 44CAD04 of Gson.

JSOUP#3676B13¹⁴: 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 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. Figure 5.12 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

¹⁴<https://github.com/jhy/jsoup/commit/3676b13>

```
Attribute o_parsesBooleanAttributes_add8698_15 = attributes.get(1);
Assert.assertEquals("boolean=\\"", ((BooleanAttribute) (o_parsesBooleanAttributes_add8
```

Figure 5.12: Test generated by DCI-*I-Amplification* that detects the behavioral change of 3676B13 of Jsoup.

`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.

```
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())
}
```

Figure 5.13: Provided test by the developer for 3676B13 of Jsoup.

MUSTACHE.JAVA#774AE7A¹⁵: 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.

```
ClasspathResolver underTest = new ClasspathResolver("templates/");
Reader reader = underTest.getReader(" does not exist");
Assert.assertNull(reader);
```

Figure 5.14: Test generated by DCI-*I-Amplification* that detects the behavioral change of 774AE7A of Mustache.java.

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 Figure 5.15. 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

¹⁵<https://github.com/spullara/mustache.java/commit/774ae7a>

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.

```
@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()));
}
```

Figure 5.15: Developer test for 774AE7A of Mustache.java.

XWIKI-COMMONS#D3101AE¹⁶: This commit fixes a bug in the `merge` method of class `DefaultDiffManager`. Figure 5.16 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 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.

```
result = this.mocker.getComponentUnderTest().merge(AmplDefaultDiffManagerTe
int o_testMergeCharList__9 = result.getLog().getLogs(LogLevel.ERROR).size()
```

Figure 5.16: Test generated by *DCI-A-Amplification* that detects the behavioral change of D3101AE of XWiki.

The developer’s test is shown in Figure 5.17. 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

¹⁶<https://github.com/xwiki/xwiki-commons/commit/d3101ae>

specified in the human test. In this case, amplified tests can be complementary.

```

@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());
}

```

Figure 5.17: Developer test for D3101AE of XWiki.

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.

5.5 Limitations

Time consumption From the experiments, it is deducible that the time consumption to complete the amplification is the main limitation of DCI. JSOUP#2C4E79B, almost 5 hours have been spent with no result. For the sake of the experimentation, a pre-defined number of iteration has been chosen to bound the exploration. In practice, setting a time budget would be recommended (*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.

Table 5.5: Standard deviations of the number of amplified tests obtained for each seed.

seed	1	2	3	4	5	6	7	8	9	10
std	63.38	63.55	62.56	61.27	61.33	61.66	63.76	60.91	61.25	63.35

False positive The risk of false positives is a potential limitation of the 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. While this is not a proof that DCI would never produce such confusing test methods, It encourages to be confident in the soundness of the implementation.

5.6 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.

For the evaluation of the randomness, a Kruskal-Willis test has been used, which is known to be weaker than ANOVA test. To perform an ANOVA test, the data must fulfil the following criteria: 1) The samples are independent; 2) Each sample is from a normally distributed population; 3) The population standard deviations of the groups are all equal. This property is known as homoscedasticity. The two first are fulfilled while the third is not: Since the standard deviations are not all equal, the associated p-value would not be valid. This is why a Kruskal-Willis test has been chosen.

In addition to this, only 11 seeds has been used to perform it, which a small samples.

However, the seeds used in DSpot's algorithm seems to not have an impact on the overall results.

5.7 Conclusion

CHAPTER 6

Transversal Contributions

[Danglot 2018] [Vera-Pérez 2018] [Yu 2019]

CHAPTER 7

Conclusion

- [Anand 2013] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil Mcminnet al. *An orchestrated survey of methodologies for automated software test case generation*. Journal of Systems and Software, vol. 86, no. 8, pages 1978–2001, 2013. (Cited on page 13.)
- [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 page 10.)
- [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 page 10.)
- [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 30.)
- [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 page 10.)
- [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 11.)
- [Cooper 1998] Harris M Cooper. Synthesizing research: A guide for literature reviews, volume 2. Sage, 1998. (Cited on page 11.)
- [Danglot 2017] Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Martin Monperrus and Benoit Baudry. *The Emerging Field of Test Amplification: A Survey*. CoRR, vol. abs/1705.10692, 2017. (Cited on page 9.)

- [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 page 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 51.)
- [Danglot 2019b] 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 27 and 57.)
- [Daniel 2009] 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 page 54.)
- [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.)
- [Duvall 2007] Paul M Duvall, Steve Matyas and Andrew Glover. Continuous integration: improving software quality and reducing risk. Pearson Education, 2007. (Cited on page 52.)
- [Edvardsson 1999] Jon Edvardsson. *A survey on automatic test data generation*. In Proceedings of the 2nd Conference on Computer Science and Engineering, pages 21–28, 1999. (Cited on page 13.)
- [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 57.)
- [Fowler 2006] Martin Fowler and Matthew Foemmel. *Continuous integration*. Thought-Works <https://www.thoughtworks.com/continuous-integration>, vol. 122, page 14, 2006. (Cited on page 52.)
- [Fraser 2011] 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 page 2.)

-
- [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.)
- [Hamlet 1993] Dick Hamlet and Jeff Voas. *Faults on its sleeve: amplifying software reliability testing*. ACM SIGSOFT Software Engineering Notes, vol. 18, no. 3, pages 89–98, 1993. (Cited on pages 11 and 12.)
- [Harrold 2008] Mary J Harrold and Alessandro Orso. *Retesting software during development and maintenance*. In Frontiers of Software Maintenance, 2008. FoSM 2008., pages 99–108, 2008. (Cited on page 13.)
- [Hetzel 1988] William C Hetzel. The complete guide to software testing. QED Information Sciences, Inc., Wellesley, MA, USA, 2nd édition, 1988. (Cited on page 9.)
- [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 52.)
- [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 10.)
- [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 58.)
- [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 10, 11 and 12.)
- [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 11 and 12.)

- [Kitchenham 2004] Barbara Kitchenham. *Procedures for performing systematic reviews*. Technical report, Keele University, 2004. (Cited on page 11.)
- [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 11 and 12.)
- [Madeyski 2010] Lech Madeyski. Test-driven development: An empirical evaluation of agile practice. Springer, 2010. (Cited on page 9.)
- [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 13.)
- [Meszaros 2006] Gerard Meszaros. Xunit test patterns: Refactoring test code. Prentice Hall PTR, 2006. (Cited on page 13.)
- [Moonen 2008] Leon Moonen, Arie van Deursen, Andy Zaidman and Magiel Bruntink. *On the Interplay Between Software Testing and Evolution and its Effect on Program Comprehension*. In Tom Mens and Serge Demeyer, editors, Software Evolution, pages 173–202. Springer, 2008. (Cited on page 13.)
- [Pacheco 2005] 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.)
- [Păsăreanu 2009] Corina S Păsăreanu and Willem Visser. *A survey of new trends in symbolic execution for software testing and analysis*. International Journal on Software Tools for Technology Transfer (STTT), vol. 11, no. 4, pages 339–353, 2009. (Cited on page 13.)
- [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 page 24.)
- [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 11 and 12.)
- [Roche 2013] James Roche. *Adopting DevOps Practices in Quality Assurance*. Commun. ACM, vol. 56, 2013. (Cited on page 1.)

-
- [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 55.)
- [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 19 and 29.)
- [van Deursen 2001] Arie van Deursen, Leon Moonen, Alex van den Bergh and Gerard Kok. *Refactoring test code*. In Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001), pages 92–95, 2001. (Cited on page 13.)
- [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 57 and 77.)
- [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 62.)
- [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 11.)
- [Xie 2006a] Tao Xie. *Augmenting Automatically Generated Unit-Test Suites with Regression Oracle Checking*. In Dave Thomas, editor, ECOOP 2006 – Object-Oriented Programming, pages 380–403, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. (Cited on pages 19 and 29.)
- [Xie 2006b] 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 page 49.)
- [Yoo 2012] Shin Yoo and Mark Harman. *Regression testing minimization, selection and prioritization: a survey*. Software Testing, Verification and Reliability, vol. 22, no. 2, pages 67–120, 2012. (Cited on page 13.)
- [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 77.)

- [Yusifoğlu 2015] Vahid Garousi Yusifoğlu, Yasaman Amannejad and Aysu Betin Can. *Software test-code engineering: A systematic mapping*. Information and Software Technology, vol. 58, pages 123 – 147, 2015. (Cited on page 13.)
- [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 page 10.)
- [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 page 10.)
- [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 11 and 12.)