Behavioural Change Detection

P H D T H E S I S

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	-	Univeristy 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

List of Tables

CHAPTER 1

Introduction

CHAPTER 2

Background

CHAPTER 3

State of the Art

CHAPTER 4

DSpot: A Test Amplification Tool

In this chapter, I expose the major output of this thesis. DSpot is a test amplification tool that have the ambition to improve the test suite of real projects, and the global quality of continuous integration process. DSpot achieves this by providing a set of automated procedures, and work at two majors levels:
1) it amplifies the test suite offline, and the outputted amplified test methods can be used to growth and improve in the project's test suite;
2) it amplified the test suite inside the Continuous Integration to enhance the developers' confidence in their changes, or increase the ability of the test suite to detect potential regression.

Contents

4.1 Definitions

We 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, long to be executed and use a lot of different component 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 would be probably the execution coverage, which can be measured a different level: branches, statements, instructions. It measures the proportion of the program that the test suite executes. The more it executes, the better is considered the test suite since it is likely to verify more behavior.

Amplified test suite is an existing test suite to which we add amplified test methods.

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.

4.2 Principle

DSpot is a test amplification tool. Its goal is to improve an existing test suite according to a specific test-criterion. DSpot takes as input the program, an existing test suite, and a test-criterion. The output of DSpot is a set of amplified test methods that are variants of existing test methods. When added to the existing test suite, it create an *amplified test suite*. This amplified test suite is better than the original test suite according to the test-criterion used during the amplification. For instance, one amplifies its test suite using branch coverage as test-criterion. This amplified test suite will execute more branches than the exiting test suite, *i.e.* the one without amplified test methods.

DSpot's principle is summarize in figure 4.1.

4.3 DSpot

DSpot is an automatic test improvement tool for Java unit tests. It is built upon the algorithms of Tonella [Tonella 2004] and Xie [Xie 2006b].

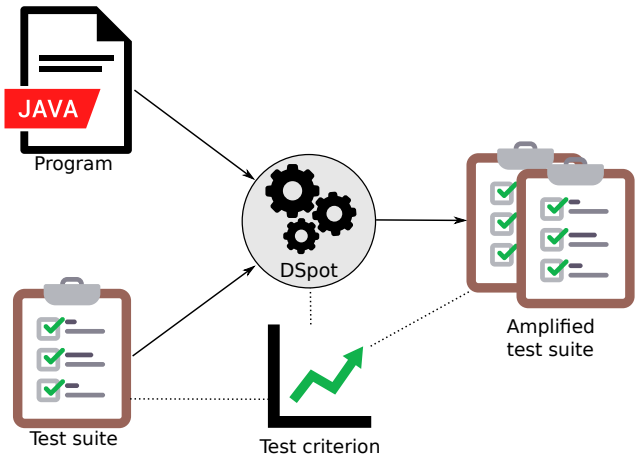

Figure 4.1: DSpot's principle: DSpot takes as input a program, an existing test suite, and a test-criterion. DSpot outputs a set of amplified test methods. When added to the existing test suite, these amplified test methods increase the test-criterion, *i.e.* the amplified test suite is better than the original one.

4.3.1 DSpot inputs

The input of DSpot consists in a set of existing test cases, manually written by the developers. As output, DSpot produces variants of the given test cases. These variants are meant to be added to the test suite. By putting together existing test cases and their variants, we aim at strictly improving the overall test suite quality. By construction, the enhanced test suite is at least as good, or better than the original one w.r.t. the considered criterion.

Concretely, DSpot synthesizes suggestions in the form of diffs that are proposed to the developer: figure 4.2 shows such a test improvement.

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

Figure 4.2: Example of what DSpot produces: a diff to improve an existing test case.

DSpot is an automatic test improvement system because it only modifies existing test cases. As such, all test improvements, by construction, are modifications to existing test cases. DSpot's novelty is twofold: 1) first, it combines those algorithms in a way that scales to modern, large Java software 2) second, it makes no assumption on the tests to be improved, and works with any arbitrary JUnit test.

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

Listing 4.1: An example of an object-oriented test case (inspired from Apache Commons Collections)

4.3.2 DSpot's Workflow

The main workflow of DSpot is composed of 2 main phases: 1) the transformation of the test code to create new test inputs inspired by Tonella's technique, we call this "input space exploration"; this phase consists in changing new test values and objects and adding new method calls, the underlying details will be explained in details in subsection 4.4.1. 2) the addition of new assertions per Xie's technique [Xie 2006b], we call this phase "assertion improvement". The behavior of the system under test is considered as the oracle of the assertion, see subsection 4.4.2. In DSpot, the combination of both techniques, *i.e.* the combination of input space exploration and assertion improvement is called "test amplification".

DSpot keeps the modifications that add the most value for the developers. To do so, DSpot uses the mutation score as a proxy to the developers' assessed value of quality. In essence, developers value changes in test code if they enable them to catch new bugs, that is if the improved test better specifies a piece of code. This is also reflected in the mutation score: if the mutation score increases, it means that a code element, say a statement, is better specified than before. In other words, DSpot uses the mutation score to steer the test case improvement, following the original conclusions of DeMillo *et al.* who observed that mutants provide hints to generate test data [DeMillo 1978]. To sum up, DSpot aims at producing better tests that have a higher potential to catch bugs.

4.3.3 Modern Test Cases

DSpot improves the test cases of modern Java programs, which are typically composed of two parts: input setup and assertions. 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.

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 written 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.

Listing 4.1 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, specifies and checks the conformance of the observed behavior with the expected one. We note that this notion of call sequence and complex objects is different from test inputs consisting only of primitive values.

4.4 Algorithms

4.4.1 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: *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, $\times 2$, $\div 2$, and replacement by an existing literal of the same type, if such literal exists. For Strings, there are four operators: add a random char, remove a random char, replace a random char and replace the string by a fully random string of the same size. 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 for 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,

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

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.

DSpot combines the different kinds of *I-Amplification* iteratively: at each iteration all kinds of *I-Amplification* are applied, resulting in new tests. From one iteration to another, DSpot reuses the previously amplified tests, and further applies *I-Amplification*.

For example, if we apply an *I-Amplification* on the example presented in Listing 4.1, it may generate a new method call on *tl*. In Listing 4.2, the added method call is "removeAll". Since DSpot changes the state of the program, existing assertions may fail. That is why it removes also all existing assertions.

4.4.2 Assertion Improvement Algorithm

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

***A-Amplification*:** *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 cases 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, the result of this execution is a map that gives, for each test case object, the values from all getters. 3) it generates new assertions in place of the observation points, using the collected values as oracle. The collected values are used as expected values in the new assertions. 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 case of the example above.

First, in Listing 4.3 DSpot instruments the test case to collect values, by addding method calls to the objects involved in the test case.

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

Listing 4.3: 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 4.4: In *A-Amplification*, the last step is to generate the assertions based on the collected values.

Second, the test with the added observation points is executed, and subsequently, DSpot generates new assertions based on the collected values. On Listing 4.4, we can see that DSpot has generated two new assertions.

4.4.3 Test Improvement Algorithm

Algorithm 1 shows the main loop of DSpot. DSpot takes as input a program P and its Test Suite TS. DSpot also uses an integer n that defines the number of iterations. DSpot produces an Amplified Test Suite ATS, *i.e.* a better version of the input Test Suite TS according to a specific test criterion such as mutation score. For each test case t in the test suite TS (Line 1), DSpot first tries to add assertions without generating any new test input (Line 3), method $generateAssertions\,(t)$ is explained in subsection 4.4.2. Note that adding missing assertions is the elementary way to improve existing tests.

DSpot initializes a temporary list of tests TMP and 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

Algorithm 1 Main amplification loop of DSpot.

Require: Program P
Require: Test Suite TS
Require: Amplifiers $amps$ to generate new test data input
Require: n number of iterations of DSpot's main loop
Ensure: An Amplified Test Suite ATS

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

subsection 4.4.1 *i.e. I-Amplification*); 2) it generates assertions on generated tests in V (Line 11 see subsection 4.4.2, *i.e. A-Amplification*); 3) it keeps the tests that improve the mutation score (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.

4.4.4 Flaky tests elimination

The input space exploration (see subsection 4.4.1) may produce test inputs that results in non-deterministic executions. This means that, between two independent executions, the state of the program is not the same. Since DSpot generates assertions where the expected value is a hard coded value from a specific run (see subsection 4.4.2), 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 generated by DSpot, we run n times each new test case resulting from amplification (n = 3 in the default configuration). If a test fails at least once, DSpot throws it away. We acknowledge that 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.

4.4.5 Selecting Focused Test Cases

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, we implement the following heuristic. 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 we value short modifications. 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, we consider that the corresponding method as specified. In that case, we do not take into account other amplified test methods that specify the same method.

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. Our goal is to select amplified tests which intent can be easily grasped by the developer: the new test specifies the method.

4.5 Implementation

DSpot is implemented in Java. It consists of 8800+ logical lines of code (as measured by cloc). For the sake of open-science, DSpot is made publicly available on Github[1]. DSpot uses Spoon[Pawlak 2015] to analyze and transform the tests of the software application under amplification.

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

CHAPTER 5

Test Improvement

Contents

5.1 Introduction

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].

In this paper we study how such modern test suites can benefit from the major results of automatic test generation research. We explore whether one can automatically improve tests written by humans, an activity that can be called "automatic test improvement". There are few works in this area: the closest related techniques are those that consider manually written tests as the starting point for an automatic test generation process

[Harder 2003, Fraser 2012, Xuan 2014, Yoo 2012, Danglot 2017, Xuan 2015]. To this extent, automatic test improvement can be seen as forming a sub-field of test generation. Automatic test improvement aims at synthesizing modifications of existing test cases, where those modifications are meant to be presented to developers. As such, the modifications must be deemed relevant by the developers themselves (the corollary being that they should not only maximize some criterion).

For our original study of automatic test improvement, we have developed DSpot, a tool for automatic test improvement in Java. DSpot adapts and combines two notable test generation techniques: evolutionary test construction [Tonella 2004] and regression oracle generation [Xie 2006a]. The essential adaptation consists in starting the generation process from the full-fledged abstract syntax trees of manually written test cases. The combination of both techniques is essential so that changes in the setup together are captured by changes in the assertion part of tests.

Our study considers 10 mature Java open source projects. It focuses on three points that have little, or never, been assessed. First, we propose 19 test improvements generated by DSpot to the developers of the considered open source projects. We present them the improvement in the form of pull requests, and we ask them whether they would like to merge the test improvements in the main repository. In this part of the study, we extensively discuss their feedback, to help the research community understand the nature of good test improvements. This reveals the key role of case studies, as presented by Flyvberg [Flyvbjerg 2006], to assess the relevance of our technique for developers. Second, we perform a quantitative assessment of the improvements of 40 real-world test classes from our set of 10 open-source projects. In particular, we consider the difficult case of improving strong test classes. Third, we explore the relative contribution of evolutionary test construction and of assertion generation in the improvement process.

Our key results are as follows: first, thirteen GitHub pull requests consisting of automatic test improvements have been definitively accepted by the developers; second, an interesting empirical fact is that DSpot has been able to improve a test class with a 99% initial mutation score (*i.e.* a really strong test); and finally, our experiment shows that valuable test improvements can be obtained within minutes.

To sum up, our contributions are:

- DSpot, a system that performs automatic test improvement of Java unit tests;
- the design and execution of an experiment to assess the relevance of automatically improved tests, 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 code and data: both DSpot[1] and our experimental data are made publicly available for future research[2]

The remainder of this article is as follows. Section **??** presents the main concepts of automatic test improvement and DSpot. Section 5.2 presents the experimental protocol of our study. Section 5.3 analyses our empirical results. Section 6.6 discusses the threats to validity. Section 5.5 discusses the related work. and Section 6.8 concludes the article. Note that a previous version of this paper can be found as Arxiv's working paper [Baudry 2015].

In this paper, we aim at improving the mutation score of test classes. In DSpot, we use Pitest[3] because: 1) it targets Java programs, 2) it is mature and well-regarded, 3) it has an active community.

An 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 case against the mutation score of the amplified version and even compare the absolute number of mutants killed by both test case variants. We will exploit this feature in our evaluation.

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.

5.2 Experimental Protocol

Automatic test improvement has been evaluated with respect to evolutionary test inputs [Tonella 2004] and new assertions [Xie 2006b]. However: 1) the two topics have never been studied in conjunction 2) they have never been studied on large modern Java programs 3) most importantly, the quality of improved tests has never been assessed by developers.

We 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, we have proposed improved tests to developers under the form of pull-requests.

We answer 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?

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

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

[3] We use the latest version released: 1.2.0.https://github.com/hcoles/pitest/releases/tag/1.2.0

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?

5.2.1 Dataset

We evaluate DSpot by amplifying test classes of large-scale, notable, open-source projects. We include projects that fulfill 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 5.3.1.

Table 5.1: Dataset of 10 active Github projects considered on our relevance study (RQ1) and quantitative experiments (RQ2, RQ3).

project	description	# LOC	# PR	considered test classes
javapoet	Java source file generator	3150	93	TypeNameTest^h NameAllocatorTest^h FieldSpecTest^l ParameterSpecTest^l
mybatis-3	Object-relational mapping framework	20683	288	MetaClassTest^h ParameterExpressionTest^h WrongNamespacesTest^l WrongMapperTest^l
traccar	Server for GPS tracking devices	32648	373	GeolocationProviderTest^h MiscFormatterTest^h ObdDecoderTest^l At2000ProtocolDecoderTest^l
stream-lib	Library for summarizing data in streams	4767	21	TestLookup3Hash^h TestDoublyLinkedList^h TestICardinality^l TestMurmurHash^l
mustache.java	Web application templating system	3166	11	ArraysIndexesTest^h ClasspathResolverTest^h ConcurrencyTest^l AbstractClassTest^l
twilio-java	Library for communicating 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

We implement those criteria as a query on top of TravisTorrent [Beller 2017]. We randomly selected 10 projects from the result of the query which produces, the dataset presented in table 5.1. 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.

5.2.2 Test Case Selection Process for Test-suite Improvement

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

First, we select unit-test classes only, because our approach focuses on unit test amplification. We use the following heuristic to discriminate unit test cases from others: we keep a test class if it executes less than an arbitrary threshold of N statements, *i.e.* if it covers a small portion of the code. In our experiment, we use $N = 1500$ based on our initial pilot experiments.

Among the unit-tests, we select 4 classes as follows. Since we want to analyze the performance of DSpot when it is provided with both good and bad tests, we select two groups of classes: one group with strong tests, one other group with low quality tests. We use the mutation score to distinguish between good and bad test classes. Accordingly, our selection process has five steps: 1) we compute the original mutation score of each class with Pitest (see section 5.1; 2) we 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 our dataset are really well-tested projects); 3) we sort the classes by mutation score (see subsection 5.2.3), in ascending order; 4) we split the set of test classes into two groups: high mutation score ($> 50\%$) and low mutation score ($< 50\%$); 5) we randomly select 2 test classes in each group.

This selection results with 40 test classes: 24 in high mutation group score and 16 in low mutation score group. The imbalance is due to the fact that there are three projects really well tested for which there are none or a single test class with a low mutation score (projects protostuff, jsoup, retrofit). Consequently, those three projects are represented with 3 or 4 well-tested classes (and 1 or 0 poorly-tested class). In table 5.1, 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.

5.2.3 Metrics

We use the following metrics during our experiment.

Number of Killed Mutants ($\#Killed.Mutants$): is the absolute number of mutants killed by a test class. We use it 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.

5.2.4 Methodology

Our experimental protocol is designed to study to what extent the test improvements are valuable for the developer.

- **RQ1** To answer to RQ1, we create pull-request on notable open-source projects. We automatically improve 19 test classes of real world applications and propose one test improvement to the main developers of each project under consideration. We propose the improvement as a pull request 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. We 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, we compute the number of suggested improvements, to verify that the developer is not overwhelmed with suggestions. We compute the number of focused amplified test cases, per the technique described in subsection 4.4.5, for each project in the benchmark. We present and discuss the proportion of focused tests out of all proposed amplified tests.

- **RQ3** To answer RQ3, we see whether the value that is taken as proxy to the developer value – the mutation score – is appropriately improved. For 40 real-world classes,

we first run the mutation testing tool Pitest (see section 5.1) on the test class. This gives the number of killed mutants for this original class. Then, we amplify the test class under consideration and we compute the new number of killed mutants after amplification. Finally, we compare and analyze the results.

- **RQ4** To answer RQ4, we compute the number of *A-Amplification* and *I-Amplification* amplifications. 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, we collect three series of metrics: 1) we compute number of killed mutants for the original test class; 2) we improve the test class under consideration using only *A-Amplification* and compute the new number of killed mutants after amplification; 3) we 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, we compare the increase of mutation score obtained by using *A-Amplification* only and *I-Amplification* + *A-Amplification*.[4]

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 our approach to select "focused" test methods, which are likely to be selected by the developers (described in subsection 4.4.5). 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 we are to find good candidates for the developers.

5.3 Experimental Results

We first discuss how automated test improvements done by DSpot are received by developers of notable open-source projects (RQ1). Then, RQ2, RQ3 and RQ4 are based on a large scale quantitative experiments over 40 real-world test classes, whose main results are reported in table 5.5. For the sake of open-science, all experimental results are made publicly available online:
`https://github.com/STAMP-project/dspot-experiments/`.

5.3.1 Answer to RQ1

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

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

5.3.1.1 Process

In this research question, our 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 we aim at providing improved tests which are easy to review. As shown in subsection 4.4.1, DSpot generates several amplified test cases, and we cannot propose them all to the developers. To select the new test case to be proposed as a pull request, we 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.

The selection of amplified test methods is done as described in subsection 4.4.5. For each selected method, we compute and minimize the diff between the original method and the amplified one and then we submit the diff as a pull request. 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, we make the diff as concise as possible for the review to be fast and easy.

5.3.1.2 Overview

In total, we have created 19 pull requests, as shown in table 5.2. 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. (This number might change over time if pull-requests are merged or closed.)

Overall 13 over 19 have been merged. Only 1 has been rejected by developers. There are 5 under discussion. In the following, we perform a manual analysis of one pull-request per project. table 5.3.1.2 contains the URLs of pull requests proposed in this experimentation.

We now present one case study per project of our dataset.

Table 5.2: 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	0	1	1
prostostuff	2	2	0	0
logback	2	0	0	2
retrofit	0	0	0	0
total	19	13	1	5

Table 5.3: 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

5.3.1.3 javapoet

We have applied DSpot 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, we have proposed to the Javapoet developers the following one liner pull request [5]:

```
181         assertThat(a.hashCode()).isEqualTo(b.hashCode());
182 +       assertFalse(a.equals(null));
```

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

5.3.1.4 mybatis-3

In project mybatis-3, We have applied DSpot 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.

We 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.*" [6]:

```
68  +   assertEquals("richType.richProperty", meta.findProperty("richType.richProperty", false));
69  +
70      assertFalse(meta.hasGetter("[0]"));
```

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

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

5.3.1.5 traccar

We have applied DSpot to amplify `ObdDecoderTest`. It identifies a single assertion that kills 14 more mutants. All newly killed mutants are located between lines 60 to 80, *i.e.* in the method `decodesCodes()` of `ObdDecoder`, which is responsible to decode a `String`. In this case, the pull request consists of a new test method because the new assertions do not fit with the intent of existing tests. This new test method is proposed into `ObdDecoderTest`, which is the class under amplification. The PR was entitled "*Improve test cases on ObdDecoder*" with the following description: "*Hello, I open this pull request to specify the method decodeCodes of the ObdDecoder*". [7]

```
17        }
18
19  +     @Test
20  +     public void testDecodeCodes() throws Exception {
21  +         Assert.assertEquals("P0D14", ObdDecoder.decodeCodes("0D14").getValue());
22  +         Assert.assertEquals("dtcs", ObdDecoder.decodeCodes("0D14").getKey());
23  +     }
24  +
25    }
```

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

5.3.1.6 stream-lib

We have applied DSpot to amplify `TestMurmurHash`. It identifies a new test input that kills 15 more mutants. All newly killed mutants are located in method `hash64()` of `MurmurHash` from lines 158 to 216. This method computes a hash for a given array of 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.*". [8]:

[7] https://github.com/tananaev/traccar/pull/2897
[8] https://github.com/addthis/stream-lib/pull/127/files

```
-         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 an new pull request with a new test method instead of changing the existing test method. We proposed, 6 days later, a second pull request entitled "*add test for hash() and hash64() against hard coded values*" with no description, since we estimated that the developer was aware of our intention.[9]:

```
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.

5.3.1.7 mustache.java

We have applied DSpot 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

[9] https://github.com/addthis/stream-lib/pull/128/files

the input of the proposed changes. We 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.*".[10]

```
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.[11].

5.3.1.8 twilio-java

We have applied DSpot to amplify `RequestTest`. It identifies two new assertions that kill 4 more mutants. All mutants were created 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*" [12]:

[10] https://github.com/spullara/mustache.java/pull/186/files

[11] the diff is same:https://github.com/spullara/mustache.java/commit/9efa19d595f893527ff218683e70db2ae4d8fb2d

[12] 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.

5.3.1.9 jsoup

We have applied DSpot 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.*"[13]:

```
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, we 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

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

or comparing the values of a few objects created for one explicit test case. The different responses we 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.

5.3.1.10 protostuff

We have applied DSpot 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. We 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.*"[14], shown earlier in figure 4.2

```
146          ByteArrayOutputStream out = new ByteArrayOutputStream();
     -       writeListTo(out, foos, SerializableObjects.foo.cachedSchema());
147  +       final int bytesWritten = writeListTo(out, foos, SerializableObjects.foo.cachedSchema());
148  +       assertEquals(0, bytesWritten);
149          byte[] data = out.toByteArray();
150
151          ByteArrayInputStream in = new ByteArrayInputStream(data);
```

A developer accepted the proposed changes the same day.

5.3.1.11 logback

We have applied DSpot 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*".[15]:

[14] https://github.com/protostuff/protostuff/pull/212/files
[15] 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.

5.3.1.12 retrofit

We 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.

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

In table 5.4, we summarize 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.

Table 5.4: 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

Most one-liner pull-requests contain both a new assertion and a new input. Consider the following Javapoet's one liner `assertFalse(x.equals(null))` (javapoet). In this example, although there is a single line starting with "assert", there is indeed a new input, the value "null".

RQ1: Would developers be ready to permanently accept improved test cases into the test repository?
Answer: We have proposed 19 test improvements 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 our 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.

5.3.2 Answer to RQ2

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

table 5.5 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

Table 5.5: 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	12	50%	19	8	599	715 19% ↗	599	0.0% →	11.11
2	NameAllocatorTest	11	87%	0	0	79	79 0.0% →	79	0.0% →	4.76
3	MetaClassTest	7	58%	108	10	455	534 17% ↗	455	0.0% →	235.71
4	ParameterExpressionTest	14	91%	2	2	162	164 1% ↗	162	0.0% →	25.93
5	ObdDecoderTest	1	80%	9	2	51	54 5% ↗	51	0.0% →	2.20
6	MiscFormatterTest	1	72%	5	5	42	47 11% ↗	42	0.0% →	1.21
7	TestLookup3Hash	2	95%	0	0	464	464 0.0% →	464	0.0% →	6.76
8	TestDoublyLinkedList	7	92%	1	1	104	105 0.97% ↗	104	0.0% →	3.03
9	ArraysIndexesTest	1	53%	15	4	576	647 12% ↗	586	1% ↗	10.58
10	ClasspathResolverTest	10	67%	0	0	50	50 0.0% →	50	0.0% →	4.18
11	RequestTest	17	81%	4	3	141	156 10% ↗	141	0.0% →	60.55
12	PrefixedCollapsibleMapTest	4	96%	0	0	54	54 0.0% →	54	0.0% →	13.28
13	TokenQueueTest	6	69%	18	6	152	165 8% ↗	152	0.0% →	15.61
14	CharacterReaderTest	19	79%	71	9	309	336 8% ↗	309	0.0% →	57.06
15	TailDelimiterTest	10	71%	1	1	381	384 0.79% ↗	381	0.0% →	12.90
16	LinkBufferTest	3	48%	12	7	66	90 36% ↗	66	0.0% →	3.24
17	FileNamePatternTest	12	58%	27	9	573	686 19% ↗	573	0.0% →	25.08
18	SyslogAppenderBaseTest	1	95%	1	1	143	148 3% ↗	143	0.0% →	7.88
19	RequestBuilderAndroidTest	2	99%	0	0	513	513 0.0% →	513	0.0% →	0.04
20	CallAdapterTest	4	94%	0	0	55	55 0.0% →	55	0.0% →	7.30
	Low mutation score									
21	FieldSpecTest	2	31%	12	4	223	316 41% ↗	223	0.0% →	4.44
22	ParameterSpecTest	2	32%	11	5	214	293 36% ↗	214	0.0% →	3.66
23	WrongNamespacesTest	2	8%	6	1	78	249 219% ↗	249	219% ↗	29.70
24	WrongMapperTest	1	8%	3	1	97	325 235% ↗	325	235% ↗	7.13
25	ProgressProtocolDecoderTest	1	16%	2	1	18	27 50% ↗	23	27% ↗	1.30
26	IgnitionEventHandlerTest	1	22%	0	0	13	13 0.0% →	13	0.0% →	0.77
27	TestICardinality	2	7%	0	0	19	19 0.0% →	19	0.0% →	2.13
28	TestMurmurHash	2	17%	40	2	52	275 428% ↗	174	234% ↗	2.18
29	ConcurrencyTest	2	28%	2	0	210	342 62% ↗	210	0.0% →	315.56
30	AbstractClassTest	2	34%	28	4	383	475 24% ↗	405	5% ↗	12.67
31	AllTimeTest	3	42%	0	0	163	163 0.0% →	163	0.0% →	0.02
32	DailyTest	3	42%	0	0	163	163 0.0% →	163	0.0% →	0.02
33	AttributeTest	2	36%	33	11	178	225 26% ↗	180	1% ↗	10.76
34	AttributesTest	5	52%	9	6	316	322 1% ↗	316	0.0% →	6.21
35	CodedDataInputTest	1	1%	0	0	5	5 0.0% →	5	0.0% →	3.58
36	CodedInputTest	1	27%	29	28	108	166 53% ↗	108	0.0% →	0.88
37	FileAppenderResilience_AS_ROOT_Test	1	4%	0	0	4	4 0.0% →	4	0.0% →	0.65
38	Basic	1	10%	0	0	6	6 0.0% →	6	0.0% →	0.89
39	ExecutorCallAdapterFactoryTest	7	62%	0	0	119	119 0.0% →	119	0.0% →	0.09
40	CallTest	35	69%	3	1	642	644 0.32% ↗	642	0.0% →	52.84

amplified test methods that met the criteria explained in subsection 4.4.5. The seventh, eight and ninth are respectively the number of killed mutants of the original test class, the number of killed mutants of its amplified version and the absolute increase obtained with amplification, which is represented with a pictogram indicating the presence of improvement. The tenth and eleventh columns concern the number of killed mutants when only A-amplification is used. The twelfth is the time consumed by DSpot to amplify the considered test class. The upper part of the table is dedicated to test classes that have a high mutation score and the lower for the test classes that have low mutation score.

For RQ2, the considered results are in the sixth column of table 5.5. Our 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 our 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 subsection 4.4.5. 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.

5.3.3 Answer to RQ3

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

In 26 out of 40 cases, DSpot is able to amplify existing test cases and improves the mutation score (MS) of the original test class. For example, let us consider the first row, corresponding to `TypeNameTest`. This test class originally includes 12 test methods that kill 599 mutants. The improved, amplified version of this test class kills 715 mutants, *i.e.* 116 new mutants are killed. This corresponds to an increase of 19% in the number of killed mutants.

We first 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 5.5. 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 well-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 5.5 (the well-tested classes), we now focus 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 we keep only test cases 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 test cases.

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 we consider the tests in the lower part of table 5.5. 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 well tested 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, we 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, we find test classes that provide a good seed for amplification. In the latter case, we have 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 cases 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 our novel 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 5.5): the mutation score of three classes is increased by more than 200%.

The most notable point of this experiment is that we have considered tests that are already really strong (table 5.5), 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.

5.3.4 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 5.5. 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, if we aggregate over all test classes, our 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.

We 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 5.5.

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

5.4 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 we propose pull requests to colleagues, they are more likely to merge them. However, this is not the case here. In this evaluation, the pull requests are submitted by the first author, who is unknown to all considered projects. The developers who study the DSpot pull requests are independent from our group and social network. Since the first author is 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, our findings would not hold for serious software projects. However, we only consider real-world applications, maintained by professional and esteemed open-source developers. This means we tried to automatically improve tests that 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 our results are of computational nature, a bug in our implementation or experimental scripts may threaten our findings. We have put all our code publicly-available for other researchers to reproduce our 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 2006a], 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.

5.5 Related Work

This work on test amplification contributes to the field of genetic improvement (GI) [Petke 2017]. The key novelty is to consider a test suite as the object to be improved, while previous GI works improve the application code. (Yet, they use the test suite as a fitness function while assessing the degree of improvement.) The work of Arcuri and Yao [Arcuri 2008] and Wilkerson *et al.* [Wilkerson 2010] are good examples of such work that use the test suite as fitness, while improving the program for automatic bug fixing. Both work follow a similar approach: evolve the input program into new versions that pass the regression test suite and that also pass the bug revealing test case (that fails on the original program). In this paper, we do not evolve the application code but the test code.

Evosuite is a state of the art tool to generate test cases for Java program [Fraser 2013]. Evosuite and DSpot have different goals. Evosuite generates new tests, while DSpot improves existing developer-written tests. The interaction between developers and synthesized tests is key here: in 2016, an empirical study demonstrated that developers who are asked to add oracles in test cases generated by Evosuite, produce test suites that are not better than manually written test suites at detecting bugs [Fraser 2015]. On the contrary, DSpot is designed to improve manually written test suites to detect more bugs, and the relevance study of RQ1 demonstrates that the outcome of DSpot is considered as valuable by developers in order to improve existing test suites.

Our work is related to previous work that aim at automatically generating test cases to improve the mutation score of a test suite. Liu *et al.* [h. Liu 2006] aim at generating small test cases, by targeting a path that covers multiple mutants to create test inputs. They evaluate their approach on five small projects. Fraser and Arcuri [Fraser 2014] propose a search-based approach to generate test suites that maximize the mutation score. However their work is different from ours since they generate new test cases from scratch, while DSpot always starts from developper-written tests. Baudry *et al.* [Baudry 2005] improve the mutation score of test suites using a bacteriological algorithm. They run experiments on a small dataset and confirm that their approach is able to increase the mutation score of tests. However, the scope of the study is limited to small programs, and they do not consider the synthesis of assertions.

Other works aim at increasing fault detection capacities of test suites. Zhang *et al.* [Zhang 2016], propose the Isomorphic Regression Testing system and its implementation in ISON. It considers two versions of a program P and P'(for instance P' is the updated version of P, on which we want to detect any regression). First, ISON identifies isomorphims, that is to say, code fragments that have the same behavior. Then, they run the test suite on P and P' to identify which of the branches are uncovered in the isomorphic part, and they collect the output. In order to cover all branches, they compute a branch condition to

execute the uncovered code. They compare ISON to Evosuite, and conclude that Evosuite achieves a better branch coverage, while ISON is able to detect faults that Evosuite does not.

Harder *et al.* [Harder 2003] start from an existing test suite. They evaluate the quality of this initial test suite with respect to operational abstractions, i.e., an abstract description of the behavior covered by the test suite. Their work is about selecting new valuable tests, while ours is about synthesizing new valuable tests.

Then, they generate novel test cases and keep only the ones that change the operational abstraction. The new test cases are generated by mining invariants using Daikon. They evaluate their approach on 8 C programs, and show that it generates test cases with good fault-detection capabilities.

Milani *et al.* [Milani Fard 2014] propose an approach which combines the advantages of manually written tests and automatic test generation. They exploit the knowledge of existing tests and then combine it with the power of automated crawling. It has been shown that the approach can effectively improve the fault detection rate of the original test suite. Test amplification, as considered in this work, is different, as it aims at enhancing the fault detection power of manually written test suites.

Yoo *et al.* [Yoo 2012] propose Test Data Regeneration(TDR), which is a kind of test amplification. They use hill climbing on existing test data (set of input) that meets a test objective (*e.g.* cover all branch of a function). The algorithm is based on *neighborhood* and a *fitness* functions as the classical hill climbing algorithm. The goal is to create new test data inputs, that have the same behavior as the original one (*e.g.* cover same branches). The key novelties of our work with respect to the work of Yoo *et al.* [Yoo 2012] are as follow: they mutate only literals in existing test cases, while DSpot's *I-Amplification* also amplifies method calls and can synthesize new objects when needed, *A-Amplification* makes the synthesis of assertions an integral part of our test suite improvement process and we evaluate the relevance of the synthesized test cases by proposing them to the developers.

Xie [Xie 2006a] proposes a technique to add assertions into existing test methods. His approach is similar to what we propose with *A-Amplification*. However, this work does not consider the synthesis of new test inputs (*I-Amplification*) and hence cannot cover new execution paths. This is the novelty of DSpot and our experiments showed that this is an essential mechanism to improve the test suite.

We now discuss a group of papers together. Pezzè *et al.* [PezzÃ¨ 2013] synthesize integration test cases from unit test cases. The idea is to combine unit test cases, which test simple functionalities on specific objects, to create new integration test cases supported by the fact that unit test cases are early developed, and integration test cases require more effort to do so. Röβler et al. [Röβler 2012] aim to isolate failure causes. They propose BugEx, a system that starts from a single failing test as input and generates test cases. It

extracts the differences in path execution between failing and passing tests. They evaluate BugEx on 7 failures and show that it is able to lead to the failure root causes in 6 cases. Yu *et al.* [Yu 2013] augment test suites to enhance fault localization. They use test input transformations to generate new test cases in existing test suites. They transform iteratively some existing failing tests to derive new test cases potentially useful to localize the specific encountered fault, similarly at *I-Amplification*. Their tool is designed to target GUI applications. To reproduce a crash occurred in production, Xuan *et al.* [Xuan 2015] propose to transform existing test cases. The approach first selects relevant test cases based on the stack trace in the crash, followed by the elimination of assertions in selected test cases, and finally uses a set of predefined transformations to produce new test cases that can help to reproduce the crash. None of those works have evaluated whether the technique scales on object-oriented applications of the size considered here, and whether the synthesized tests are considered valuable by senior developers.

It can be noted that several test generation techniques start from a seed and evolve it to produce a good test suite. This is the case for techniques such as concolic test generation [Godefroid 2005], search-based test generation [Fraser 2012], or random test generation [Groce 2007]. The main difference between all these works and DSpot lies in the nature of the seed: previous work use input values in the form of numerical or String values, vectors or files, and do not consider any form of oracle. On the contrary, we consider as a seed a real test case. It means the seed is a complete program, which creates objects, manipulates the state of these objects, calls methods on these objects and asserts properties on their behavior. This is the contribution of DSpot: using real and complex object-oriented tests as seed.

Almasi *et al.* [Almasi 2017] investigate the efficiency and effectiveness of automated test generation on a production ready application named *LifeCalc*. They use 25 real faults from *LifeCalc* to evaluate two state-of-the-art tools, Evosuite and Randoop, by asking feedback from the developers about the generated test methods. The result are as follows: overall the tools found 19 over 25 real faults; The developers state that the assertions and the readability of generated test methods must be improved. The developers also suggest that such tools should be implemented in continuous integration. The reason of the 7 faults that remain undetected is that they either require complex test data input or specific assertions. Our experiment is larger in scope, we evaluate DSpot on 10 notable open-source software from GitHub, by proposing amplified test methods in pull requests.

Allamanis *et al.* [Allamanis 2014] devised a technique to rename elements in code and evaluate their approach through five pull requests where four of them have been accepted. Their work and ours both rely on independent evaluation through pull-requests. One important difference is that, in the description of the pull request, they say that the improvements are generated by a tool, while in our case, we did not say anything about the

research project underlying our pull requests.

5.6 Conclusion

We have presented DSpot, a novel approach to automatically improve existing developer-written test classes. We have shown that DSpot is able to strengthen real unit test classes in Java from 10 real-world projects. Our experiment with real developers indicates that they are ready to merge test cases improved by DSpot into their test suite. The road ahead for automatic synthesis of test case improvements is exciting.

First, there is a need to study how to generate meaningful natural language explanations of the suggested test improvements: generation of well named tests, generation of text accompanying the pull request, we dream of using natural-language deep-learning for this task.

Second, we aim at automating even more the process of integrating the amplification result in a ready-to-use pull request. This requires two major steps: first, one needs to identify which parts of the amplified test methods are "valuable". Second, we need to choose between modifying an existing test method or create a new one that is derived from an existing one, even if the new method is by construction an extension of an existing one. Such a decision procedure must be made based on the intention of the existing test methods and the potentially new intention of the amplified test. If we find an existing test method that carries the same intention, *i.e.* it tests the same portion of code as the amplification, one would preferably add changes there rather than creating a new test methods. This challenging vision of mining and comparing test purposes is the main area of our future work.

Third, and finally, we envision to integrate DSpot in a continuous integration service (CI) where test classes would be amplified on-the-fly for each commit. This would greatly improve the direct industrial applicability of this software engineering research.

CHAPTER 6

Behavioural Change Detection for Commit

When a developer pushes a change to an application's codebase, in the form of a commit, the newly introduced behavior may be incorrect. To prevent such regressions, developers rely on a continuous integration (CI) server to run a test suite on the application, at every commit. However, if the test suite lacks the test cases that specify the behavioral changes introduced by this commit, or the changes introduced by previous commits, the bug goes undetected.
In this paper, we propose an approach that takes a program, its test suite, and a commit as input, and generates test methods that detect the behavioral changes of the commit, i.e., the behavioral difference between the pre-commit and post-commit versions of the program. In essence, this helps developers evolve the test suite (i.e., the application's specification) at the same time they evolve the application. We call our approach DCI (Detecting behavioral changes in CI). It works by generating variations of the existing test cases through (i) assertion amplification and (ii) a search-based exploration of input spaces. We evaluate our approach on a curated set of 60 commits from 6 open source Java projects. Our study exposes the characteristics of commits in modern open-source software and the ability of our approach to generate test methods that detect behavioral changes.

Contents

6.1 Introduction

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 we address in this paper.

In this paper, we aim to automatically generate test methods for each commit that is submitted to the CI. In particular, we generate a test case t_{gen} that specifies the behavioral change of each commit. We consider a generated test case t_{gen} 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, we developed a new approach, called DCI, that works in two steps. First, we analyze the test cases of the pre-commit version and select the ones that exercise the parts of the code modified by the commit. Second, our test generation techniques produce variant test cases that either add assertions [Xie 2006b] to existing tests or explore new inputs following a search-based test input generation approach [Tonella 2004]. This process of automatic generation of t_{gen} from existing tests is called *test amplification* [Zhang 2012]. We evaluate our approach on a benchmark of 50 commits selected from 5 open source Java projects, constructed with a novel and systematic methodology. We analyzed 1510 commits and selected those that introduce a behavioral change (*e.g.*, we do not want to generate tests for commits that only change comments). We also make sure that all selected commits contain a developer-written test case that detects the behavioral change. In our protocol, the developer's test case acts as a ground-truth to analyze the tests generated by DCI. Overall, we found 50 commits that satisfy the two essential properties we are looking for: 1) the commit introduces a behavioral change; 2) the commit has a human written test we can use for ground truth.

To sum up, our contributions are:

- DCI (**D**etecting behavioral changes in **CI**), an approach based on *test amplification* to generate new tests that detect the behavioral change introduced by a commit.
- An open-source implementation of DCI for Java.
- A curated benchmark of 60 commits that introduce a behavioral change and include a test case to detect it, selected from 6 notable open source Java projects[1].
- A comprehensive evaluation based on 4 research questions that combines quantitative and qualitative analysis with manual assessment.

In Section 6.2 we motivate the need to have commits include a test case that specifies the behavioral change. In Section 6.3 we introduce our technical contribution: an approach for commit-based test selection and amplification. Section 6.4 introduces our benchmark of commits, the evaluation protocol and the results of our experiments on 50 real commits. Section 6.6 relates the threats validity and actions that have been taken to overcome them. In Section 6.7, we expose the related work, their evaluation and the differences with our work and eventually we conclude in Section 6.8.

[1] https://github.com/danglotb/DCI

6.2 Motivation & Background

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

6.2.1 Motivating Example

On August 10, a developer pushed a commit to the master branch of the XWiki-commons project. The change[2], displayed in Figure 6.1, 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.

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

Figure 6.1: Commit 7e79f77 on XWiki-Commons that changes the behavior without a test.

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, `equals` being a highly used method. Such a central behavioral change may impact the whole program, and the lack of a test case for this new behavior may have dramatic consequences in the future. Without a test case, 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.

Our work on automatic test amplification in continuous integration aims at mitigating such risk: test amplification aims at ensuring that every commit include a new test method or a modification of an existing test method. In this paper, we study how 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. Our goal is to use this new test method to ensure that any changed behavior can be caught in the future.

What we propose is as follows: 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 case to detect a behavioral change, our

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

approach would provide one. DCI would take as input the commit and a test suite, and generate a new test case that detects the behavioral change.

6.2.2 Practibility

We describe a complete scenario to sum up our vision of our approach's usage.

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 case 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 our definition, 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].

From our experience, unit tests (vs integration test) are the best target for DCI The reasons are behind the very nature of unit tests: First, they have a small scope, which allow DCI to intensify its search while an integration test, that contains a lot of code, would make DCI 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 DCI needs to execute 5 times the tests under amplification, it means that DCI would be executed faster when it amplifies unit tests than when it amplified integration tests.

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.

6.2.3 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 case 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.

6.2.4 Behavioral Change Detection

Behavioral change detection is the task of identifying or generating a test or an input that distinguishes a behavioral change between two versions of the same program. In this paper, we propose a novel approach to detect behavioral changes based on test amplification.

6.2.5 Test Amplification

Test amplification is the idea of improving existing tests with respect to a specific test criterion [Zhang 2012]. We start from an existing test suite and create variant tests that improve a given test objective. For instance, a test amplification tool may improve the code coverage of the test suite. In this paper, our test objective is to improve the test suite's detection of behavioral changes introduced by commits.

6.3 Behavioral Change Detection Approach

We propose an approach to produce test methods that detect the behavioral changes introduced by commits. We call our approach DCI (**D**etecting behavioral changes in **CI**), and propose it be used during continuous integration.

6.3.1 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 two kinds of test amplification to generate new test methods that detect the behavioral change. Figure 6.2 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.

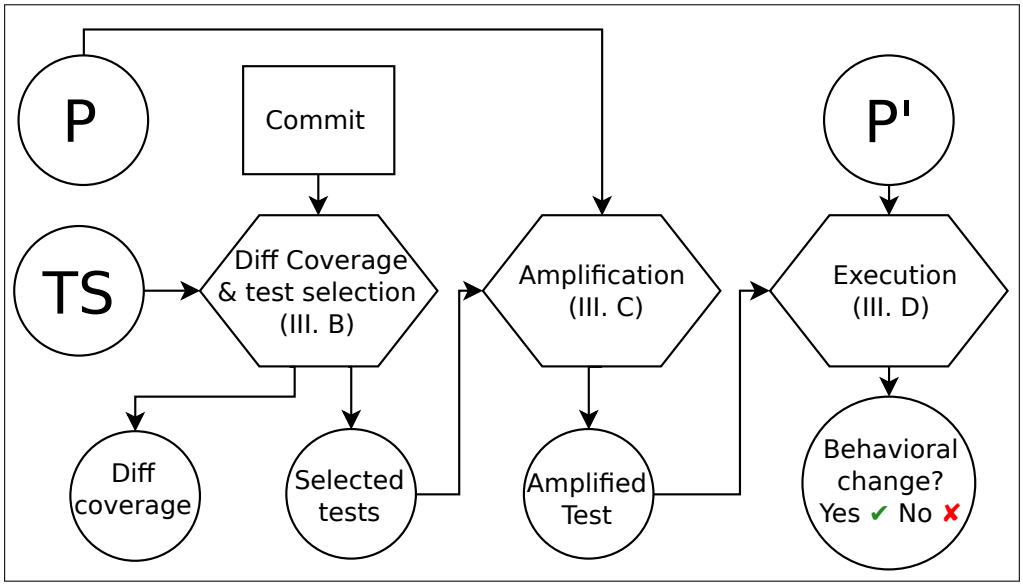

Figure 6.2: Overview of our approach to detect behavioral changes in commits.

6.3.2 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, we select only test methods that are present in pre-commit version (*i.e.*, we ignore 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.

6.3.3 Test Amplification

Once we have the initial tests that cover the diff, we want to make them detect the behavioral change and assess the new behavior. This process of extending the scope of a test case is called test amplification [Zhang 2012]. In DCI, we build upon Xie's technique [Xie 2006b] and Tonella's evolutionary algorithm [Tonella 2004] to perform test amplification.

6.3.3.1 Assertion Amplification

A test method consists of a setup and assertions. The former is responsible for putting the program under test into a specific state; the latter is responsible for verifying that the actual

state of the program at the end of the test is the expected one. To do this, assertions compare actual values against expected values: if the assertion holds, the program is considered correct, if not, the test case has revealed the presence of a bug.

Assertion amplification (AAMPL) has been proposed by [Xie 2006b]. It takes as input a program and its test suite, and it synthesizes new assertions on public methods that capture the program state. The targeted public methods are those that take no parameter, return a result, and match a Java naming convention of getters, *e.g.* the method starts with *get* or *is*. The standard method *toString()* is also used. If a method used returns a complex Java Object, AAMPL recursively uses getters on this object to generate deeper assertions.

In case the test method sets the program into an incorrect state and an exception is thrown, AAMPL generates a test for this exception by wrapping the test method body in a *try/catch* block. It also inserts a *fail* statement at the end of the body of the *try*, *i.e.* it means that if the exception is not thrown the test method fails.

Algorithm 2 AAMPL: Assertion amplification algorithm.

Require: Program P
Require: Test Suite TS
Ensure: An Amplified Test Suite ATS

```
1:  ATS ← ∅
2:  for Test in TS do
3:      NoAssertTest ← removeAssertions(Test)
4:      InstrTest ← instrument(NoAssertTest)
5:      execute(InstrTest)
6:      AmplTest ← NoAssertTest.clone()
7:      for Observ in InstrTest.observations() do
8:          Assert ← generateAssertion(Observ)
9:          AmplTest ← AmplTest.add(Assert)
10:     end for
11:     ATS.add(select(AmplTest))
12:     ATS.add(AmplTest)
13: end forreturn ATS
```

We present AAMPL's pseudo-code in Algorithm Algorithm 2. First, it initializes an empty set of tests ATS (Line 1). For each $Test$ method in the test suite TS (Line 2), it removes the existing assertions to obtain $NoAssertTest$ (Line 3). Then, it instruments $NoAssertTest$ with observation points (Line 4) that allow retrieving values from the program at runtime, which results in $InstrTest$. In order to collect the values, it executes $InstrTest$ (Line 5). Eventually, for each observation $Observ$ of the set of observations from $InstrTest$ (Line 7 to 10), it generates an assertion (Line 8) and adds it to the amplified tests $AmplTest$ (Line 9). At the end, it selects amplified test according to a specific test criterion using the method $select()$ (Line 11) and add selected amplified test methods to the set of test methods $AmplTest$, in other words, an amplified test suite (Line 13).

To sum up, AAMPL increases the number of assertions. By construction, it specifies more behaviors than the original test suite. DCI$_{AAMPL}$ is the AAMPL mode for DCI.

6.3.3.2 Search-based Amplification

Search-based test amplification consists in running stochastic transformations on test code [Tonella 2004]. For DCI$_{AAMPL}$, this process consists in a) generating a set of original test methods by applying code transformations; b) running AAMPL to synthesize new assertions for these amplified test methods; c) repeating this process nb times[3], each time seeding with the previously amplified test methods. This final step allows the search-based algorithm to explore more inputs, and thus improve the chance of triggering new behaviors.

Algorithm 3 SBAMPL: Search based amplification algorithm

Require: Program P
Require: Program P'
Require: Test Suite TS
Require: Iterations number Nb
Ensure: An Amplified Test Suite ATS

```
1:  ATS ← ∅
2:  TmpTests ← ∅
3:  for Test in TS do
4:      TmpTests ← Test
5:      for i ← 0, i < Nb do
6:          TransformedTests ← transform(TmpTests)
7:          AmplifiedTests ← aampl(TransformedTests)
8:          ATS.add(select(AmplifiedTests))
9:          TmpTests ← AmplifiedTests
10:     end for
11: end forreturn ATS
```

We present the search-based amplification algorithm in Algorithm Algorithm 3. This algorithm is a basic Hill Climbing algorithm. It takes as input a program with two distinct versions P and P', its test suite TS and a number of iterations nb, (in our case $nb = 3$). It produces an amplified test suite that contains test methods that pass on P but fail on P'. To do so, it initializes an empty set of amplified test methods ATS (Line 1), which will be the final output, and $TmpTests$ (Line 2) which is a temporary set. Then, for each test method in the test suite TS (Line 3), it applies the following operations: 1) transform the current set of test methods (Line 6) to obtain $TransformedTests$; 2) apply AAMPL on $TransformedTests$ (Line 7, see Algorithm 2) to obtain $AmplifiedTests$; 3) select amplified test methods using the method $select()$, and add them to ATS (the method $select()$ executes the amplified tests on P' and keeps only tests that fail, *i.e.* that detect

[3] by default, $nb = 3$

a behavioral change); and Finally, 4) affects $AmplifiedTests$ to $TmpTests$ in order to stack transformations.

In our study, we consider the following test transformations:

- On numbers: 1. add 1 to an integer 2. minus 1 to an integer 3. replace an integer by zero 4. replace an integer by the maximum value (Integer.MAX_VALUE in Java) 5. replace an integer by the minimum value (`Integer.MIN_VALUE` in Java).

- On booleans: 1. negate the value.

- On string literals: 1. replace a string with another existing string 2. replace a string with white space, or a system path separator, or a system file separator. 3. add 1 random character to the string 4. remove 1 random character from the string 5. replace 1 random character in the string by another random character 6. replace the string with a random string of the same size 7. replace the string with the `null` value

- On methods : 1. remove a method call 2. duplicate a method call.

DCI$_{SBAMPL}$ is the search-based amplification mode for DCI.

6.3.4 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.

6.3.5 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.[4]

[4] `https://github.com/STAMP-project/dspot.git`

6.4 Evaluation

To evaluate the DCI approach, we design an experimental protocol to answer the following research questions:

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

6.4.1 Benchmark

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

Project selection We need software projects that are 1) publicly-available, 2) written in Java, 3) and use continuous integration. We pick the projects from the dataset in [Vera-Pérez 2018] and [Danglot 2019], which is composed of mature Java projects from GitHub.

Commit selection We take commits in inverse chronological order, from newest to oldest. We select the first ten commits that match the following criteria: 1) the commit modifies Java files (most behavioral changes are source code changes.[5]); 2) the commit provides or modifies a manually written test that detects a behavioral change. To verify this property, we execute the test on the pre-commit version. If it fails, it means that the test detects at least 1 behavioral change. We will use this test 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, we compute the diff coverage. If the coverage is 0%, we discard the commit. We do this because if the change is not covered, we cannot select any test methods to be amplified, which is what we want to evaluate.

Together, these criteria ensure that all selected commits: 1) introduce behavioral changes, 2) provide or modify a manually written test case that detects a behavioral change (which will be used as ground-truth for comparing generated tests), and 3) that there is at

[5]We are aware that behavioral changes can be introduced in other ways, such as modifying dependencies or configuration files [Hilton 2018].

least 1 test in the pre-commit version of the program that 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 6.1: 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 6.1 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 we did the analysis; 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 we analyze, the total number of commits we discard, the number of commits that match all our criteria but the third (there is no test in the pre-commit that execute the change and the number of commit we select). We note that our benchmark is only composed of recent commits from notable open-source projects and is available on GitHub at `https://github.com/STAMP-project/dspot-experiments`.

6.4.2 Protocol

To answer **RQ1**, we run DCI$_{AAMPL}$ and DCI$_{SBAMPL}$ on the benchmark projects. We then report the total number of behavioral changes successfully detected by DCI, *i.e.* 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. We also discuss 1 case study of a successful behavioral change detection.

To answer **RQ2**, we run DCI$_{SBAMPL}$ for 1, 2 and 3 iterations on the benchmark projects. We report the number of behavioral changes successfully detected for each number of iterations in the main loop. We report 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 We perform a Kruskal-Wallis test statistic on these numbers.

The null hypothesis is the following: The population median of all of the groups are equal

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

We take a confidence level of 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, we perform a manual analysis.We randomly select 1 commit per project in the benchmark, and we manually analyze whether the automatically selected tests for this commit are semantically related to the behavioral changes in the commit.

To answer **RQ4**, we use the ground-truth tests written or modified by developers in the selected commits. We manually compare the amplified test methods that detect behavioral changes to the human tests, for 1 commit per project.

6.4.3 Results

The overall results are reported in Table 6.2. 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$_{AAMPL}$, 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 the eleventh are respectively a ✓ with the number of amplified tests for DCI$_{SBAMPL}$ (or - if a change is not detected) for 3 iterations.

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

In this section, we describe the the characteristics of commits introducing behavioral changes in the context of continuous integration The first five columns in Table 6.2 describe the characteristics of our benchmark. The commit dates show that the benchmark is only composed of recent commits. The most recent is GSON#B1FB9CA, authored 9/22/18, and the oldest is COMMONS-IO#5D072EF, authored 9/10/15. The number of test methods at the time of the commit shows two aspects of our benchmark: 1) we only have strongly tested projects; 2) we see that the number of tests evolve over time due to test evolution. Every commit in the benchmark comes with test modifications (new tests

Table 6.2: Performance evaluation of DCI on 60 commits from 6 large open-source projects.

	id	date	#Test	#Modified Tests	+ / -	Cov	#Selected Tests	#AAMPL Tests	Time	#SBAMPL Tests	Time
commons-io	c6b8a38	6/12/18	1348	2	104 / 3	100.0	3	0	10.0s	0	98.0s
	2736b6f	12/21/17	1343	2	164 / 1	1.79	8	0	19.0s	✓ (12)	76.3m
	a4705cc	4/29/18	1328	1	37 / 0	100.0	2	0	10.0s	0	38.1m
	f00d97a	5/2/17	1316	10	244 / 25	100.0	2	✓ (1)	10.0s	✓ (39)	27.0s
	3378280	4/25/17	1309	2	5 / 5	100.0	1	✓ (1)	9.0s	✓ (11)	24.0s
	703228a	12/2/16	1309	1	6 / 0	50.0	8	0	19.0s	0	71.0m
	a7bd568	9/24/16	1163	1	91 / 83	50.0	8	0	20.0s	0	65.2m
	81210eb	6/2/16	1160	1	10 / 2	100.0	1	0	8.0s	✓ (8)	23.0s
	57f493a	11/19/15	1153	1	15 / 1	100.0	8	0	7.0s	0	54.0s
	5d072ef	9/10/15	1125	12	74 / 34	68.42	25	✓ (6)	29.0s	✓ (1538)	2.2h
	total						66	0.80	avg(14.5s)	160.80	avg(38.8m)
commons-lang	f56931c	7/2/18	4105	1	30 / 4	25.0	42	0	2.4m	0	8.6h
	87937b2	5/22/18	4101	1	114 / 0	77.78	16	0	35.0s	0	18.1m
	09ef69c	5/18/18	4100	1	10 / 1	100.0	4	0	16.0s	0	98.8m
	3fadfdd	5/10/18	4089	1	7 / 1	100.0	9	0	17.0s	✓ (4)	17.2m
	e7d16c2	5/9/18	4088	1	13 / 1	33.33	7	0	16.0s	✓ (2)	15.1m
	50ce8c4	3/8/18	4084	4	40 / 1	90.91	2	✓ (1)	28.0s	✓ (135)	2.0m
	2e9f3a8	2/11/18	4084	2	79 / 4	30.0	47	0	79.0s	0	66.5m
	c8e61af	2/10/18	4082	1	8 / 1	100.0	10	0	17.0s	0	16.0s
	d8ec011	11/12/17	4074	1	11 / 1	100.0	5	0	31.0s	0	2.3m
	7d061e3	11/22/17	4073	1	16 / 1	100.0	8	0	17.0s	0	11.4m
	total						150	0.10	avg(40.5s)	14.10	avg(74.7m)
gson	b1fb9ca	9/22/17	1035	1	23 / 0	50.0	166	0	4.2m	0	92.5m
	7a9fd59	9/18/17	1033	2	21 / 2	83.33	14	0	15.0s	✓ (108)	2.1m
	03a72e7	8/1/17	1031	2	43 / 11	68.75	371	0	7.7m	0	3.2h
	74e3711	6/20/17	1029	1	68 / 5	8.0	1	0	4.0s	0	16.0s
	ada597e	5/31/17	1029	2	28 / 3	100.0	5	0	8.0s	0	8.7m
	a300148	5/31/17	1027	7	103 / 2	18.18	665	0	9.2m	✓ (6)	4.9h
	9a24219	4/19/17	1019	1	13 / 1	100.0	36	0	2.2m	0	48.9m
	9e6f2ba	2/16/17	1018	2	56 / 2	50.0	9	0	32.0s	✓ (2)	8.5m
	44cad04	11/26/16	1015	1	6 / 0	100.0	2	0	15.0s	✓ (37)	40.0s
	b2c00a3	6/14/16	1012	4	242 / 29	60.71	383	0	7.9m	0	3.6h
	total						1652	0.00	avg(3.2m)	15.30	avg(86.5m)
jsoup	426ffe7	5/11/18	668	4	27 / 46	64.71	27	✓ (2)	42.0s	✓ (198)	33.6m
	a810d2e	4/29/18	666	1	27 / 1	80.0	5	0	10.0s	0	26.6m
	6be19a6	4/29/18	664	1	23 / 1	50.0	50	0	69.0s	0	67.7m
	e38dfd4	4/28/18	659	1	66 / 15	90.0	18	0	35.0s	0	12.5m
	e9feec9	4/15/18	654	1	15 / 3	100.0	4	0	9.0s	0	95.0s
	0f7e0cc	4/14/18	653	2	56 / 15	84.62	330	0	6.5m	✓ (36)	11.8h
	2c4e79b	4/14/18	650	2	82 / 2	50.0	44	0	67.0s	0	4.7h
	e5210d1	12/22/17	647	1	3 / 3	100.0	14	0	9.0s	0	4.9m
	df272b7	12/22/17	647	2	17 / 1	100.0	13	0	9.0s	0	4.6m
	3676b13	12/21/17	648	6	104 / 12	38.46	239	0	6.2m	✓ (52)	6.8h
	total						744	0.20	avg(101.0s)	28.60	avg(2.6h)
mustache.java	a1197f7	1/25/18	228	1	43 / 57	77.78	131	0	11.8m	✓ (204)	10.1h
	8877027	11/19/17	227	1	22 / 2	33.33	47	0	7.3m	0	100.2m
	d8936b4	2/1/17	219	2	46 / 6	60.0	168	0	12.7m	0	84.2m
	88718bc	1/25/17	216	2	29 / 1	100.0	1	✓ (1)	7.0s	✓ (149)	3.7m
	339161f	9/23/16	214	2	32 / 10	77.78	123	0	8.6m	✓ (1312)	5.8h
	774ae7a	8/10/16	214	2	17 / 2	100.0	11	0	66.0s	✓ (124)	6.8m
	94847cc	7/29/16	214	2	17 / 2	100.0	95	0	11.5m	✓ (2509)	21.4h
	eca08ca	7/14/16	212	4	47 / 10	80.0	18	0	87.0s	0	41.8m
	6d7225c	7/7/16	212	2	42 / 4	80.0	18	0	87.0s	0	40.1m
	8ac71b7	7/6/16	210	10	167 / 31	40.0	20	0	2.1m	✓ (124)	5.6m
	total						632	0.10	avg(5.8m)	442.20	avg(4.2h)
xwiki-commons	ffc3997	7/27/18	1081	0	125 / 18	21.05	1	0	29.0s	0	18.0s
	ced2635	8/13/18	1081	1	21 / 14	60.0	5	0	93.0s	0	2.5h
	10841b1	8/1/18	1061	1	107 / 19	30.0	51	0	5.7m	0	3.4h
	848c984	7/6/18	1074	1	154 / 111	17.65	1	0	28.0s	0	18.0s
	adfefec	6/27/18	1073	1	17 / 14	40.0	22	✓ (1)	76.0s	✓ (3)	14.9m
	d3101ae	1/18/18	1062	2	71 / 9	20.0	4	✓ (1)	72.0s	✓ (31)	41.4m
	a0e8b77	1/18/18	1062	2	51 / 8	42.86	4	✓ (1)	72.0s	✓ (60)	42.1m
	78ff099	12/19/17	1061	1	16 / 0	33.33	2	0	68.0s	✓ (4)	6.6m
	1b79714	11/13/17	1060	1	20 / 5	60.0	22	0	78.0s	0	17.9m
	6dc9059	10/31/17	1060	1	4 / 14	88.89	22	0	79.0s	0	20.5m
	total						134	0.30	avg(94.3s)	9.80	avg(49.5m)
	total						3378	9(15)	avg(2.2m)	28(6708)	avg(109.4m)

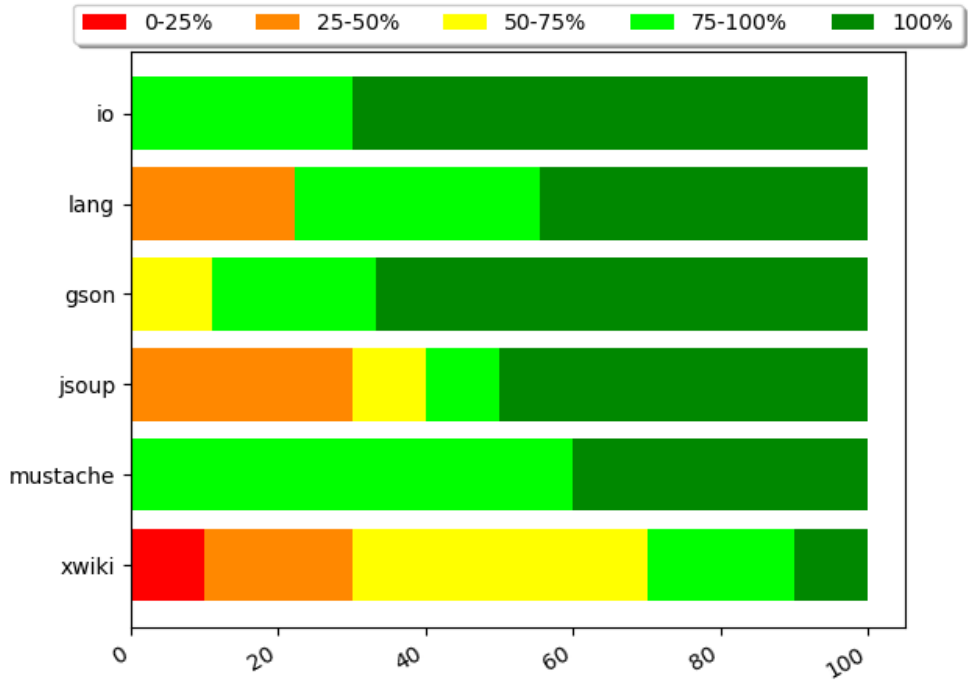

Figure 6.3: Distribution of diff coverage per project of our benchmark.

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 6.3: 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 our selection criteria that ensures the presence of a test specifying the behavioral change.

Thanks to our selection criteria, we have a curated benchmark of 50 commits with a behavioral change, coming from notable open-source projects, and covering a diversity of commit sizes. The benchmark is publicly available and documented for future research on this topic.

6.4.3.2 RQ1: To what extent are DCI$_{AAMPL}$ and DCI$_{SBAMPL}$ able to produce amplified test methods that detect the behavioral changes?

We now focus on the last 4 columns of Table 6.2. For instance, for COMMONS-IO#F00D97A (4^{th} row), DCI$_{AAMPL}$ generated 39 amplified tests that detect the behavioral change. For COMMONS-IO#81210EB (8^{th} row), only the SBAMPL version of DCI de-

tects the change. Overall, using only AAMPL, DCI generates amplified tests that detect 9 out of 60 behavioral changes. Meanwhile, using SBAMPL only, DCI generates amplified tests that detect 28 out of 60 behavioral changes.

Regarding the number of generated tests. DCI_{SBAMPL} generates a large number of test cases, compared to DCI_{AAMPL} only (15 versus 6708, see column "total" at the bottom of the table). Both DCI_{AAMPL} and DCI_{AAMPL} can generate amplified tests, however since DCI_{AAMPL} 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_{AAMPL} only adds assertions, the amplified tests are easier to understand than the ones generated by DCI_{SBAMPL}.

DCI_{SBAMPL} takes more time than DCI_{AAMPL} (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_{SBAMPL}, while DCI_{AAMPL} 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(28 out of 60). Recall that the 60 commits that we analyze are real changes that fix bugs in complex code bases. They represent modifications, sometimes deep in the code, that represent challenges with respect to testability [Voas 1995]. Consequently, the fact DCI can generate 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.

Now, we manually analyze a successful case where DCI detects the behavioral change. We select commit 3FADFDD[6] from commons-lang, which is succinct enough to be discussed in the paper. The diff is shown in Figure 6.4.

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

Figure 6.4: Diff of commit 3FADFDD from commons-lang.

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

DCI starts the amplification from the `testNestingPerson` test method defined in `JsonToStringStyleTest`. The test is selected for amplification because it triggers the execution of the changed line.

We show in Figure 6.5 the resulting amplified test method. From this test method, DCI generates an amplified test method shown in Figure 6.5. 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

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

```
ToStringBuilder o_testNestingPerson_add33752__20 =
    new ToStringBuilder(nestP).append("pid", nestP.pid).append("per/on",
Assert.assertEquals("{\"pid\":\"#1@Jane\",\"per/on\":{\"name\":\"Jane Doe
```

Figure 6.5: Test generated by DCI that detects the behavioral change of 3FADFDD from commons-lang.

an 's' inside "person" by '/' resulting in "per/on" where "/" is a special character that must be escaped (Line 2). Then, DCI generated 11 assertions, based on the modified inputs. The amplified test the behavioral change: in the pre-commit version, the expected value is: `"{ ... 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 is capable of detecting the behavioral changes in a total of 28/60 commits. Individually, DCI_{SBAMPL} finds changes in 28/60, while DCI_{AAMPL} in 9/60 commits. Since DCI_{SBAMPL} also uses AAMPL to generate assertions, all DCI_{AAMPL}'s commits are contained in DCI_{SBAMPL}'s. However, the search-based algorithm, through exploration, finds many more behavioral changes, making it more effective albeit at the cost of execution time.

6.4.3.3 RQ2: What is the impact of the number of iteration performed by DCI_{SBAMPL}?

The results are reported in Table 6.3 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_{SBAMPL} 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_{SBAMPL} 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_{SBAMPL} 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 then 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_{SBAMPL} 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_{SBAMPL}

explores more with more iteration and thus is able to generate more amplified test methods that detect the behavioral changes.

In average DCI$_{SBAMPL}$ 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 6.4. The result of the Kruskal-Wallis test is: p-value=0.96. $p - value > \alpha$ which means that we keep the null hypothesis: The population median of all of the groups are equal. This means that, in general, the choice of the seeds has not a significant impact of the overall result of DCI.

Answer to **RQ2**: DCI$_{SBAMPL}$ 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$_{SBAMPL}$ impacts the number of behavioral changes detected, the number of amplified test methods obtained and the execution time.

6.4.3.4 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 our protocol described in Subsection 6.4.2, we answer this question based on manual analysis: we randomly selected 1 commit per project, and we analyzed the relevance of the selected tests for amplification.

In order to give an intuition of what we consider as a relevant test selection for amplification, let us look at an example. If `TestX` is selected for amplification, following a change to method `X`, we consider this as relevant. The key is that DCI will generate an amplified test `TestX'` that is a variant of `TestX`, and, consequently, the developer will directly get the intention of the new test `TestX'` and what behavioral change it detects.

COMMONS-IO#C6B8A38[7]: our 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`. Two test methods out 3 (`testCopyURLToFileWithTimeout` and `testCopyURLToFile`) there is a link between the changed file and the intention of tests to be amplified. The selection is thus considered relevant.

COMMONS-LANG#F56931C[8]: our test selection returns 39 test methods from 5 test classes: `FastDateFormat_ParserTest`, `FastDateParserTest`, `DateUtil-`

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

Table 6.3: Evaluation of the impact of the number of iteration done by DCI$_{SBAMPL}$ on 60 commits from 6 open-source projects.

	id	$it = 1$	Time	$it = 2$	Time	$it = 3$	Time
commons-io	c6b8a38	0	25.0s	0	62.0s	0	98.0s
	2736b6f	✓ (1)	26.1m	✓ (2)	44.2m	✓ (12)	76.3m
	a4705cc	0	4.1m	0	21.1m	0	38.1m
	f00d97a	✓ (7)	13.0s	✓ (28)	19.0s	✓ (39)	27.0s
	3378280	✓ (6)	15.0s	✓ (10)	20.0s	✓ (11)	24.0s
	703228a	0	30.3m	0	55.1m	0	71.0m
	a7bd568	0	28.6m	0	52.0m	0	65.2m
	81210eb	✓ (2)	14.0s	✓ (4)	18.0s	✓ (8)	23.0s
	57f493a	0	20.0s	0	32.0s	0	54.0s
	5d072ef	✓ (461)	32.2m	✓ (1014)	65.5m	✓ (1538)	2.2h
	total	47.70	avg(12.3m)	105.80	avg(24.0m)	160.80	avg(38.8m)
commons-lang	f56931c	0	0.0s	✓ (6)	9.8h	0	8.6h
	87937b2	0	3.5m	0	10.5m	0	18.1m
	09ef69c	0	97.0s	0	21.0m	0	98.8m
	3fadfdd	✓ (1)	2.0m	✓ (1)	9.3m	✓ (4)	17.2m
	e7d16c2	✓ (3)	111.0s	✓ (2)	8.4m	✓ (2)	15.1m
	50ce8c4	✓ (61)	38.0s	✓ (97)	78.0s	✓ (135)	2.0m
	2e9f3a8	0	11.4m	0	35.0m	0	66.5m
	c8e61af	0	16.0s	0	16.0s	0	16.0s
	d8ec011	0	36.0s	0	68.0s	0	2.3m
	7d061e3	0	79.0s	0	5.8m	0	11.4m
	total	6.50	avg(2.3m)	10.60	avg(68.2m)	14.10	avg(74.7m)
gson	b1fb9ca	0	14.6m	0	51.0m	0	92.5m
	7a9fd59	✓ (7)	33.0s	✓ (48)	73.0s	✓ (108)	2.1m
	03a72e7	0	30.2m	0	102.3m	0	3.2h
	74e3711	0	6.0s	0	11.0s	0	16.0s
	ada597e	0	61.0s	0	4.9m	0	8.7m
	a300148	0	45.2m	✓ (4)	2.6h	✓ (6)	4.9h
	9a24219	0	10.8m	0	28.4m	0	48.9m
	9e6f2ba	0	79.0s	0	4.5m	✓ (2)	8.5m
	44cad04	✓ (4)	21.0s	✓ (21)	30.0s	✓ (37)	40.0s
	b2c00a3	0	31.5m	0	111.8m	0	3.6h
	total	1.10	avg(13.6m)	7.30	avg(46.0m)	15.30	avg(86.5m)
jsoup	426ffe7	✓ (126)	5.4m	✓ (172)	19.2m	✓ (198)	33.6m
	a810d2e	0	90.0s	0	13.9m	0	26.6m
	6be19a6	0	8.1m	0	39.7m	0	67.7m
	e38dfd4	0	117.0s	0	6.3m	0	12.5m
	e9feec9	0	20.0s	0	50.0s	0	95.0s
	0f7e0cc	0	2.4h	✓ (7)	6.8h	✓ (36)	11.8h
	2c4e79b	0	7.1m	0	34.1m	0	4.7h
	e5210d1	0	45.0s	0	2.3m	0	4.9m
	df272b7	0	43.0s	0	2.2m	0	4.6m
	3676b13	✓ (6)	21.4m	✓ (35)	2.8h	✓ (52)	3.3h
	total	13.30	avg(19.4m)	21.40	avg(69.5m)	28.60	avg(2.2h)
mustache.java	a1197f7	✓ (28)	5.9h	✓ (124)	8.4h	✓ (204)	10.1h
	8877027	0	30.5m	0	58.4m	0	100.2m
	d8936b4	0	3.2m	0	4.8m	0	84.2m
	88718bc	✓ (13)	78.0s	✓ (85)	2.5m	✓ (149)	3.7m
	339161f	✓ (143)	115.9m	✓ (699)	4.1h	✓ (1312)	5.8h
	774ae7a	✓ (18)	2.7m	✓ (65)	4.7m	✓ (124)	6.8m
	94847cc	✓ (122)	5.3h	✓ (580)	10.4h	✓ (2509)	21.4h
	eca08ca	0	8.1m	0	24.3m	0	41.8m
	6d7225c	0	7.9m	0	26.8m	0	40.1m
	8ac71b7	✓ (2)	2.7m	✓ (48)	3.8m	✓ (124)	5.6m
	total	32.60	avg(84.3m)	160.10	avg(2.5h)	442.20	avg(4.2h)
	ffc3997	0	19.0s	0	18.0s	0	18.0s
	ced2635	0	8.0m	0	31.8m	0	2.5h

Table 6.4: 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 DCI$_{AAMPL}$ and DCI$_{SBAMPL}$ able to produce amplified test methods that detect the behavioral changes?. The ten following columns are the result obtained for the 10 different seeds.

	id	0	1	2	3	4	5	6	7	8	9
commons-io	c6b8a38	-	-	-	-	-	-	-	-	-	-
	2736b6f	1	1	-	2	1	1	2	1	-	-
	a4705cc	-	-	-	-	-	-	-	-	-	-
	f00d97a	7	7	7	7	7	7	7	7	7	7
	3378280	6	6	6	6	6	6	6	6	6	6
	703228a	-	-	-	-	-	2	-	1	-	-
	a7bd568	-	-	-	-	-	2	-	1	-	-
	81210eb	2	2	2	2	2	2	2	2	2	2
	57f493a	-	-	-	-	-	-	-	-	-	-
	5d072ef	464	462	463	463	461	462	465	460	463	462
commons-lang	f56931c	-	-	-	-	1	-	1	1	1	1
	87937b2	-	-	-	-	-	-	-	-	-	-
	09ef69c	-	-	-	-	-	-	-	-	-	-
	3fadfdd	1	-	2	3	3	2	1	-	-	1
	e7d16c2	-	1	-	-	1	1	-	1	1	1
	50ce8c4	61	63	62	57	58	60	61	62	63	61
	2e9f3a8	-	-	-	-	-	-	-	-	-	-
	c8e61af	-	-	-	-	-	-	-	-	-	-
	d8ec011	-	-	-	-	-	-	-	-	-	-
	7d061e3	-	-	-	-	-	-	-	-	-	-
gson	b1fb9ca	-	-	-	-	-	-	-	-	-	-
	7a9fd59	7	7	7	7	7	7	7	7	7	7
	03a72e7	-	-	-	-	-	-	-	-	0	3
	74e3711	-	-	-	-	-	-	-	-	-	-
	ada597e	-	-	-	-	-	-	-	-	-	-
	a300148	-	-	-	-	-	-	-	-	-	-
	9a24219	-	-	-	-	-	-	-	-	-	-
	9e6f2ba	-	-	-	-	-	-	-	-	-	-
	44cad04	5	4	6	4	4	4	4	4	3	5
	b2c00a3	-	-	-	-	-	-	-	-	-	-
jsoup	426ffe7	125	125	123	126	126	127	123	121	119	121
	a810d2e	-	-	-	-	-	-	-	-	-	-
	6be19a6	-	-	-	-	-	-	-	-	-	-
	e38dfd4	-	-	-	-	-	-	-	-	-	-
	e9feec9	-	-	-	-	-	1	-	1	-	-
	0f7e0cc	-	-	1	1	-	1	1	-	1	-
	2c4e79b	-	-	-	-	-	-	-	-	-	-
	e5210d1	-	-	-	-	-	-	-	-	-	-
	df272b7	-	-	-	-	-	-	-	-	-	-
	3676b13	3	6	2	3	4	1	1	1	1	-
mustache.java	a1197f7	22	-	-	-	-	-	-	-	-	-
	8877027	-	-	-	-	-	-	-	-	-	-
	d8936b4	-	-	-	-	-	-	-	-	-	-
	88718bc	12	11	13	13	11	13	12	12	12	12
	339161f	135	146	-	-	-	-	138	-	-	144
	774ae7a	23	18	3	17	18	15	16	19	17	20
	94847cc	-	-	112	-	61	72	57	40	42	37
	eca08ca	-	-	-	-	-	-	-	-	-	-
	6d7225c	-	-	-	-	-	-	-	-	-	-
	8ac71b7	2	2	2	2	2	2	2	2	2	2
xwiki-commons	ffc3997	-	-	-	-	-	-	-	-	-	-
	ced2635	-	-	-	-	-	-	-	-	-	-
	10841b1	-	-	-	-	-	-	-	-	-	-
	848c984	-	-	-	-	-	-	-	-	-	-
	adfefec	1	1	1	1	1	1	1	1	1	1
	d3101ae	18	15	13	14	15	14	-	15	12	15
	a0e8b77	19	-	14	15	17	15	16	16	13	16
	78ff099	4	4	4	4	4	4	4	4	4	4
	1b79714	-	-	-	-	-	-	-	-	-	-
	6dc9059	-	-	-	-	-	-	-	-	-	-

sTest, FastDateParser_TimeZoneStrategyTest and FastDateParser_MoreOrLessTest. This commit modifies the behavior of two methods: simpleQuote and setCalendar of class FastDateParser. Our manual analysis 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. It is considered as an half-success.

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

JSOUP#E9FEEC9[10], our 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 manual test added in the commit is also in the XmlTreeBuilderTest class. If DCI proposes a new test there to capture the behavioral change, the developer will understand its relevance and its relation to the change.

MUSTACHE.JAVA#88718BC[11], our test selection returns the testInvalidDelimiters test method defined in the com.github.mustachejava.InterpreterTest test class. The commit improves an error message when an invalid delimiter is used. Here, the test selection is relevant since it selected testInvalidDelimiters which is the dedicated test to the usage of the test invalid delimiters. This ground-truth test method is also in the test class com.github.mustachejava.InterpreterTest.

XWIKI-COMMONS#848C984[12] our 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, this is considered as a success.

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, we selected over and above the tests to be amplified. That is, we select tests whose intention is semantically pertinent to the change, but we also include tests that are not. However, even in this case, DCI's test selection provides developers with

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

important and targeted context to better understand the behavioral change at hand.

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

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

COMMONS-IO#81210EB[13]: This commit modifies the behavior of the `read()` method in `BoundedReader`. Figure 6.6 shows the test generated by DCI$_{SBAMPL}$. 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$_{SBAMPL}$ 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 amplication.

```
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 6.6: Test generated by DCI$_{SBAMPL}$ that detects the behavioral change introduced by commit 81210EB in commons-io.

Now, let us look at the human test contained in the commit, shown in Figure 6.7. 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`.

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

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

COMMONS-LANG#E7D16C2[14]: this commit escapes special characters before adding them to a `StringBuffer`. Figure 6.8 shows the amplified test method obtained by DCI$_{SBAMPL}$. 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 DCI test fails since the code now escapes the special character \.

```
  new ToStringBuilder(this.base).append("a", '\n').append("b", 'B');
Assert.assertEquals("{\"a\":\"\n\",\"b\":\"B\",",
  ((StringBuffer) o_testChar_add45986__10.getStringBuffer()).toString())
```

Figure 6.8: Test generated by DCI$_{SBAMPL}$ that detects the behavioral change of E7D16C2 in commons-lang.

Let's have a look to the human test method shown in Figure 6.9. 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.

GSON#44CAD04[15]: This commit allows Gson to deserialize a number represented as a string. Figure 6.10 shows the relevant part of the test generated by DCI$_{SBAMPL}$, based on `testNumberDeserialization` of `PrimitiveTest` as a seed. First, we see

[14] https://github.com/apache/commons-lang/commit/e7d16c2
[15] https://github.com/google/gson/commit/44cad04

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

Figure 6.9: Developer test for E7D16C2 of commons-lang.

that the test selected as a seed is indeed related to the change in the deserialization feature. The DCI test detects the behavioral change 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). [16].

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

Figure 6.10: Test generated by DCI that detects the behavioral change of commit 44CAD04 in Gson.

We compare it against the developer-written ground-truth method, shown in Figure 6.11. 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.

JSOUP#3676B13[17]: 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 6.12

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

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

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

Figure 6.11: Provided test by the developer for 44CAD04 of Gson.

shows an amplified test obtained using DCI$_{SBAMPL}$. This amplified test has 15 assertions and a duplication of method call. Thanks to this duplication and assertion generated on the `toString()` method, this test is able to capture the behavioral change introduced by the commit.

```
Attribute o_parsesBooleanAttributes_add8698__15 = attributes.get(1);
Assert.assertEquals("boolean=\"\"", ((BooleanAttribute) (o_parsesBooleanAttributes_add8
```

Figure 6.12: Test generated by DCI$_{SBAMPL}$ that detects the behavioral change of 3676B13 of Jsoup.

As before, we compare it to the developer's test. The developer uses the `Element` and `outerHtml()` methods rather than `Attribute` and `toString()`. However, the method `outerHtml()` in `Element` will call the `toString()` method of `Attribute`. For this behavioral change, it concerns the `Attribute` and not the `Element`. So, the amplified test is arguably better, since it is closer to the change than the developer's test. But, DCI$_{SBAMPL}$ 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 6.13: Provided test by the developer for 3676B13 of Jsoup.

MUSTACHE.JAVA#774AE7A[18]: 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

[18] https://github.com/spullara/mustache.java/commit/774ae7a

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 6.14: Test generated by DCI$_{SBAMPL}$ 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 6.15. These tests invoke the method `getReader` which is the modified method in the commit. The difference is that the DCI$_{SBAMPL}$'s amplified test method provides a non longer valid input for the method `getReader`. However, providing such inputs produce errors afterward which signal the behavioral change. In this case, the amplified test is complementary to the human test since it verifies that the wrong inputs are no longer supported and that the system immediately throws an error.

```
@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 6.15: Developer test for 774AE7A of Mustache.java.

XWIKI-COMMONS#D3101AE[19]: This commit fixes a bug in the `merge` method of class `DefaultDiffManager`. Figure 6.16 shows the amplified test method obtained by DCI$_{AAMPL}$. 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

[19] https://github.com/xwiki/xwiki-commons/commit/d3101ae

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 6.16: Test generated by DCI$_{AAMPL}$ that detects the behavioral change of D3101AE of XWiki.

The developer's test is shown in Figure 6.17. 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$_{AAMPL}$ and makes the human test clearly of better quality but also longer to write. Yet, DCI$_{AAMPL}$'s amplified tests capture a behavioral change that was not 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.g

    // 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 6.17: Developer test for D3101AE of XWiki.

Answer to **RQ4**: In 2 of 5 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.

6.5 Discussion about the scope of DCI

In this section, we overview the current limitations of DCI and the key factors that prevent DCI from amplifying test methods that detect behavioral changes.

Time consumption From our experiments, we see 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 our experimentation, we choose to use a pre-defined number of iteration to bound the exploration. In practice, we recommend to set a time budget (*e.g.* at most one hour per pull-request).

Importance of test seeds By construction, DCI's effectiveness is correlated to the test methods used as seed. For example, see the row of `commons-lang#c8e61af` in Table 6.3 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.

False positive The risk of false positives is a potential limitation of your 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. We 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, we are confident in the soundness of our implementation.

6.6 Threats to validity

An internal threat is the potential bugs in the implementation of DCI. However, we heavily tested our prototype with JUnit test cases to mitigate this threat.

In our benchmark, there are 60 commits. Our result may be not be generalizable to all programs. But we carefully selected real and diverse applications from GitHub, all having a strong test suite. We believe that the benchmark reflects real programs, and we have good confidence in the results.

Last but not least threat is the potential flakiness of generated test methods. However we take care that our approach does not produce flaky test methods, and we make sure to observe a stable and different state of the program between different executions. To do this, we execute 3 times each amplified test in order to check weather or not there are stable. If the outcome of a at least one execution is different than the others, we discard the amplified test.

For the evaluation of the randomness, we use a Kruskal-Willis, which is known to be weaker than ANOVA test. To perform an ANOVA test, the data must fulfull the following

Table 6.5: 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

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 we choose to use a Kruskal-Willis test.

In addition to this, we perform it for only 11 seeds, which a small samples.

6.7 Related Work

6.7.1 Commit-based test generation

Person *et al.* [Person 2008] present differential symbolic execution (DSE). DSE combines symbolic execution and a new approximation technique to highlight behavioral changes. They use symbolic execution summary to find equivalences and difference and generate a set of inputs that trigger different behavior. This is done in three steps: 1) they execute both versions of the modified method; 2) they find equivalences and differences, thanks to the analysis of symbolic execution summary; 3) they generate a set of inputs that trigger the different behaviors in both versions. The main difference with our work is that they have the strong assumption to have a program whose semantics is fully handled by the symbolic execution engine. In the context of Java, to our knowledge, no symbolic execution engine works on arbitrary Java program. Symbolic execution engines do not scale to the size and complexity of the programs we targeted. On the contrary, our approach, being more lightweight, is meant to work on all Java programs.

Marinescu and Cadar [Marinescu 2013] present Katch, a system that aims at covering the code included in a patch. This approach first determines the differences of a program and its previous version. It targets modified and not executed by the existing test suite lines. Then, it selects the closest input to each target from existing tests using a static minimum distance over the control flow graph. The proposal is evaluated on Unix tools. They examine patches from a period of 3 years. In average, they automatically increase coverage from 35% to 52% with respect to the manually written test suite. Contrary to our work, they only aim at increasing the coverage, not at detecting behavioral changes.

A posterior work of the same group [Palikareva 2016, Kuchta 2018] focuses on finding test inputs that execute different behaviors in two program versions. They devise a technique, named ShaddowKlee, built on top of Klee [Cadar 2008]. They require the code to be

annotated at changed places. Then they select from the test suite those test cases that cover the changed code. The unified program is used in a two stage dynamic symbolic execution guided by the selected test cases. They first look for branch points where the conditions are evaluated in both program versions. Then, a constraint solver generates new test inputs for divergent scenarios. The program versions are then normally executed with the generated inputs and the result is validated to check the presence of a bug or of an intended difference. The evaluation of the proposed method is based on the CoREBench [Böhme 2014] data set that contains documented regression bugs of the GNU Coreutils program suite.

Noller *et al.* [Noller 2018] aim at detecting regression bugs. They apply shadow symbolic execution, originally from Palikevera [Person 2008, Palikareva 2016] that has been discussed in the previous paragraph, on Java programs. Their approach has been implemented as an extension of Java Path Finder Symbolic (jpf-symbc)[Anand 2007], named jpf-shadow. Shadow symbolic execution generate test inputs that trigger the new program behavior. They use a merged version of both version of the same program, i.e. the previous version, so called old, and the changed version, called new. This is done by instrumenting the code with method calls "change()". The method change() takes two inputs: the old statement and the new one. Then, a first step collects divergence points, i.e. conditional where the old version and the new version do not take the same branch. On small examples, they show that jpf-shadow generates less unit test cases yet cover the same number of path. Jpf-shadow only aims at covering the changes and not at detecting the behavioral change with an assertion.

Menarini *et al.* [Menarini 2017] proposes a tool, GETTY, based on invariants mined by Daikon. GETTY provides to code reviewers a summary of the behavioral changes, based on the difference of invariants for various combinations of programs and test suites. They evaluate GETTY on 6 open source project, and showed that their behavioral change summaries can detect bugs earlier than with normal code review. While they provide a summary, DCI provides a concrete test method with assertions that detect the behavioral changes.

Lahiri *et al.* [Lahiri 2013] propose differential assertion checking (DAC): checking two versions of a program with respect to a set of assertions. DAC is based on filtering false alarms of verification analysis. They evaluate DAC on a set of small example. The main difference is that DAC requires to manually write specifications, while DCI is completely automated with normal code as input.

Yang *et al.* [Yang 2014] introduce IProperty, a way to annotate correctness properties of programs. They evaluate their approach on the triangle problem. The key novelty of our work is to perform an evaluation on real commits from large scale open source software.

Campos *et al.* [Campos 2014] extended EvoSuite to adapt test generation techniques to continuous integration. Their contribution is the design of a time budget allocation

strategy: it allocates more time budget to specific classes that are involved in the changes. They evaluated their approach on 10 projects from the SF100 corpus, on 8 of the most popular open-source projects from GitHub, and on 5 industrial projects. They limit their evaluation to the 100 last consecutive commits. They observe an increase of +58% branch coverage, +69% thrown undeclared exceptions, while reducing the time consumption by up to 83% compared to the baseline. The major difference compared to our approach, they do not aim at specifically obtaining test methods that detect the behavioral changes but rather obtain better branch coverage and detect undeclared exceptions. They also do not generate any assertions. However, from the point of view practitioners, integrating a time budget strategy into DCI would increase its usability, practicability and potential adoption.

6.7.2 Behavioral change detection

Evans *et al.* [Evans 2007] devise the differential testing. This approach aims at alleviating the test repair problem and detects more changes than regression testing alone. They use an automated characterization test generator (ACTG) to generate test suite for both version of the program. They then categorizes the tests of these 2 test suites into 3 groups: 1) $T_{preserved}$ which are the tests that pass on the both versions; 2) $T_{regressed}$ which are the tests that pass on the previous version but not on the new one; 3) $T_{progressed}$ which are the tests that pass on the new version but not on the previous one; Then, they define also $T_{different}$ which is the union of both $T_{regressed}$ and $T_{progressed}$. The approach is to execute $T_{different}$ on both versions and observe progressed and regressed behaviors. They evaluate their approach on a small use case from the SIR dataset on 38 diffrent changes, for version of the program. They showed that their approach detects 21%, 34%, and 21% more behavior changes than regression testing alone for respectively version 1, version 2 and version 3. In DCI, the amplified test methods obtained would lie into the $T_{regressed}$ group. However, we could also amplified test methods using the new version of the program and obtain a $T_{progressed}$. We would obtain a $T_{different}$ of amplified test methods and it might improve the performance of DCI. About the evaluation, we run experimentation of 60 commits which the double than their dataset, and on real projects and real commits from GitHub.

Wei Jin *et al.* [Jin 2010] propose BEhavioral Regression Testing BERT. BERT aims at assisting practitioners during development to identify potential regression. It has been implemented as a plugin for the IDE Eclipse. Each time a developer make a change in their code base and Eclipse compiles, BERT is triggered. BERT works in 3 phases: 1) it analyzes what are the classes modified and runs a test generation tools, such as Randoop, to create new test input for these classes. 2) it executes the generated tests on both version of the program and collect multiples values such as the values of the fields of objects, the returned values by methods, etc. 3) it produces a report containing all the differences of behaviors based on the collected values. Then the developer used this report to decide

whether or not the changes are correct. They evaluated BERT on a small and artificial project, showing that about 60% of the automatically generated test inputs were able to reveal the behavioral difference that indicates the regression fault In addition to this proof-of-concept, they evaluated in on JODA-time, which is a mature and widely used library. They evaluated on 54 pairs of versions. They reported 36 behavioral differences. However, they could establish only for one of them was a regression fault. There are two major differences with DCI: 1) DCI works at commit level and not to the class changes level. 2) DCI produces real and actionable test methods.

Taneja *et al.* [Taneja 2008] present DiffGen, a tool that generate regression tests for two version of the same class. Their approach works as follow: First, they detect the changes between the two version of the class. It is done using the textual representation and at method level. Second, they generate what they call a test driver, which is a class that contains a method for each modified method. These methods takes as input an instance of the old version of the class and the inputs required by the modified method. They also make all the field public to compare their values between the old version and the new one. These comparison have the form of branches. The intuition is if the test generator engine is able to cover these branches, it will reveal the behavioral differences. Third, they generate test using a test generator and the test driver. Eventually, they execute the generated tests to see whether or not there is a behavioral difference. They evaluated DiffGen on 8 artificial classes from the state of the art. They compared the mutation score of their generated test suite to an existing method from the state of the art. They showed that that DiffGen has an Improvement Factor IF2 varying from 23.4% to 100% for all the subjects. They also performed an evaluation on larger subjects from the SIR dataset. They detected 5 more faults than the state of the art. DiffGen must modified the application code to be efficient while DCI does not required any modification of it. Thus, is makes generated tests by DiffGen unused by developers since they must expose all the fields of their classes.

6.7.3 Test amplification

Yoo *et al.* [Yoo 2012] devise Test Data Regeneration(TDR). They use hill climbing on existing test data (set of input) that meets a test objective (*e.g.* cover all branch of a function). The algorithm is based on *neighborhood* and a *fitness* functions as the classical hill climbing algorithm. The key difference with DCI is that they at fulfilling a test criterion, such as branch coverage, while we aim at obtaining test methods that detect the behavioral changes.

It can be noted that several test generation techniques start from a seed and evolve it to produce a good test suite. This is the case for techniques such as concolic test generation [Godefroid 2005], search-based test generation [Fraser 2012], or random test generation [Groce 2007]. The key novelty of DCI relies in the very nature of the tests we used as seed.

DCI uses complete program, which creates objects, manipulates the state of these objects, calls methods on these objects and asserts properties on their behavior. That is to say real and complex object-oriented tests as seed

6.7.4 Continuous Integration

Hilton *et al.* [Hilton 2016] conduct a study on the usage, costs and benefits of CI. To do this, they use three sources: open-source code, builds from Travis, and they surveyed 442 engineers. Their studies show that the usage of CI services such as Travis is widely used and became the trend. The fact that CI is widely used shows that relevance of behavioral change detection.

Zampetti *et al.* [Zampetti 2017] investigate the usage of Automated Static Code Analysis Tools (ASCAT) in CI. There investigation is done on 20 projects on GitHub. According to their findings, coding guideline checkers are the most used static analysis tools in CI. This paper shows that dynamic analysis, such as DCI, is the next step for getting more added-value from CI.

Spieker *et al.* [Spieker 2017] elaborate a new approach for test case prioritization in continuous integration based on reinforcement learning. Test case prioritization is different from behavioral change detection.

Waller *et al.* [Waller 2015] study the portability of performance tests in continuous integration. They show little variations of performance tests between runs (every night) and claim that the performance tests must be integrated in the CI, early as possible in the development of Software. Performance testing is also one kind of dynamic analysis for the CI, but different in nature from behavioral change detection.

6.8 Conclusion

In this paper, we have studied the problem of behavioral change detection for continuous integration. We have proposed a novel technique called DCI, which uses assertion generation and search-based transformation of test code to generate tests that automatically detect behavioral changes in commits. We have evaluated our technique on a curated set of 50 commits coming from real-world, large open-source Java projects.

We plan to work on an automated continuous integration bot for behavioral change detection that will: 1) check if a behavioral change is already specified in a commit (*i.e.* a test case that correctly detects the behavioral change is provided); 2) if not, execute behavioral change detection and test generation; 3) propose the synthesized test method to the developers to complement the commit. Such a bot can work in concert with other continuous integration bots, such as bots for automated program repair [Urli 2018].

CHAPTER 7

Transversal Contributions

CHAPTER 8

Thesis Perspectives and Future Works

CHAPTER 9

Conclusion

Bibliography

[Allamanis 2014] Miltiadis Allamanis, Earl T. Barr, Christian Bird and Charles Sutton. *Learning Natural Coding Conventions*. In Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pages 281–293, New York, NY, USA, 2014. ACM. (Cited on page 43.)

[Almasi 2017] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri and J. Benefelds. *An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application*. In 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), pages 263–272, May 2017. (Cited on page 43.)

[Anand 2007] Saswat Anand, Corina S. Pasareanu and Willem Visser. *JPF-SE: A symbolic execution extension to Java pathfinder*, 03 2007. (Cited on page 74.)

[Arcuri 2008] Andrea Arcuri and Xin Yao. *A novel co-evolutionary approach to automatic software bug fixing*. In Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on, pages 162–168. IEEE, 2008. (Cited on page 41.)

[Baudry 2005] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel and Le Traon Yves. *From Genetic to Bacteriological Algorithms for Mutation-Based Testing*. Software, Testing, Verification & Reliability journal (STVR), vol. 15, no. 2, pages 73–96, June 2005. (Cited on page 41.)

[Baudry 2015] Benoit Baudry, Simon Allier, Marcelino Rodriguez-Cancio and Martin Monperrus. *DSpot: Test Amplification for Automatic Assessment of Computational Diversity*. ArXiv paper 1503.05807, 2015. (Cited on page 19.)

[Beck 2003] K. Beck. Test-driven development: by example. Addison-Wesley Professional, 2003. (Cited on page 17.)

[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 22.)

[Böhme 2014] Marcel Böhme and Abhik Roychoudhury. *Corebench: Studying complexity of regression errors*. In Proceedings of the 2014 International Symposium on Software Testing and Analysis, pages 105–115. ACM, 2014. (Cited on page 74.)

[Cadar 2008] Cristian Cadar, Daniel Dunbar and Dawson Engler. *KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs*. In Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association. (Cited on page 73.)

[Campos 2014] José Campos, Andrea Arcuri, Gordon Fraser and Rui Abreu. *Continuous Test Generation: Enhancing Continuous Integration with Automated Test Generation*. In Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, pages 55–66, New York, NY, USA, 2014. ACM. (Cited on page 74.)

[Danglot 2017] Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Martin Monperrus and Benoit Baudry. *The Emerging Field of Test Amplification: A Survey*. arXiv preprint arXiv:1705.10692, 2017. (Cited on page 18.)

[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 page 55.)

[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 49.)

[DeMillo 1978] Richard A DeMillo, Richard J Lipton and Frederick G Sayward. *Hints on test data selection: Help for the practicing programmer*. Computer, vol. 11, no. 4, pages 34–41, 1978. (Cited on page 10.)

[Duvall 2007] Paul M Duvall, Steve Matyas and Andrew Glover. Continuous integration: improving software quality and reducing risk. Pearson Education, 2007. (Cited on page 46.)

[Evans 2007] Robert B Evans and Alberto Savoia. *Differential testing: a new approach to change detection*. In The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers, pages 549–552. ACM, 2007. (Cited on page 75.)

[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 54.)

[Flyvbjerg 2006] Bent Flyvbjerg. *Five misunderstandings about case-study research*. Qualitative inquiry, vol. 12, no. 2, pages 219–245, 2006. (Cited on page 18.)

[Fowler 2006] Martin Fowler and Matthew Foemmel. *Continuous integration*. ThoughtWorks https://www.thoughtworks.com/continuous-integration, vol. 122, page 14, 2006. (Cited on page 46.)

[Fraser 2012] Gordon Fraser and Andrea Arcuri. *The seed is strong: Seeding strategies in search-based software testing*. In Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on, pages 121–130. IEEE, 2012. (Cited on pages 18, 43 and 76.)

[Fraser 2013] Gordon Fraser and Andrea Arcuri. *Whole test suite generation*. IEEE Transactions on Software Engineering, vol. 39, no. 2, pages 276–291, 2013. (Cited on page 41.)

[Fraser 2014] Gordon Fraser and Andrea Arcuri. *Achieving Scalable Mutation-based Generation of Whole Test Suites*. Empirical Software Engineering, vol. 20, no. 3, pages 783–812, 2014. (Cited on page 41.)

[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 41.)

[Godefroid 2005] Patrice Godefroid, Nils Klarlund and Koushik Sen. *DART: directed automated random testing*. In ACM Sigplan Notices, volume 40, pages 213–223. ACM, 2005. (Cited on pages 43 and 76.)

[Groce 2007] Alex Groce, Gerard Holzmann and Rajeev Joshi. *Randomized differential testing as a prelude to formal verification*. In Proceedings of the 29th international conference on Software Engineering, pages 621–631. IEEE Computer Society, 2007. (Cited on pages 43 and 76.)

[h. Liu 2006] M. h. Liu, Y. f. Gao, J. h. Shan, J. h. Liu, L. Zhang and J. s. Sun. *An Approach to Test Data Generation for Killing Multiple Mutants*. In 2006 22nd IEEE International Conference on Software Maintenance, pages 113–122, Sept 2006. (Cited on page 41.)

[Harder 2003] Michael Harder, Jeff Mellen and Michael D. Ernst. *Improving Test Suites via Operational Abstraction*. In Proc. of the Int. Conf. on Software Engineering (ICSE), pages 60–71, 2003. (Cited on pages 18 and 42.)

[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 pages 46 and 77.)

[Hilton 2018] 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 55.)

[Jin 2010] W. Jin, A. Orso and T. Xie. *Automated Behavioral Regression Testing*. In 2010 Third International Conference on Software Testing, Verification and Validation, pages 137–146, April 2010. (Cited on page 75.)

[Kuchta 2018] Tomasz Kuchta, Hristina Palikareva and Cristian Cadar. *Shadow Symbolic Execution for Testing Software Patches*. ACM Trans. Softw. Eng. Methodol., vol. 27, no. 3, pages 10:1–10:32, September 2018. (Cited on page 73.)

[Lahiri 2013] Shuvendu Lahiri, Kenneth McMillan and Chris Hawblitzel. *Differential Assertion Checking*. Technical report, March 2013. (Cited on page 74.)

[Marinescu 2013] Paul Dan Marinescu and Cristian Cadar. *KATCH: high-coverage testing of software patches*. page 235. ACM Press, 2013. (Cited on page 73.)

[Menarini 2017] M. Menarini, Y. Yan and W. G. Griswold. *Semantics-assisted code review: An efficient tool chain and a user study*. In 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 554–565, Oct 2017. (Cited on page 74.)

[Milani Fard 2014] Amin Milani Fard, Mehdi Mirzaaghaei and Ali Mesbah. *Leveraging existing tests in automated test generation for web applications*. In Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, pages 67–78. ACM, 2014. (Cited on page 42.)

[Noller 2018] Yannic Noller, Hoang Lam Nguyen, Minxing Tang and Timo Kehrer. *Shadow Symbolic Execution with Java PathFinder*. SIGSOFT Softw. Eng. Notes, vol. 42, no. 4, pages 1–5, January 2018. (Cited on page 74.)

[Palikareva 2016] Hristina Palikareva, Tomasz Kuchta and Cristian Cadar. *Shadow of a doubt: testing for divergences between software versions*. In Proceedings of the 38th International Conference on Software Engineering, pages 1181–1192. ACM, 2016. (Cited on pages 73 and 74.)

[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 15.)

[Person 2008] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum and Corina S. Păsăreanu. *Differential Symbolic Execution*. In Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '08/FSE-16, pages 226–237, New York, NY, USA, 2008. ACM. (Cited on pages 73 and 74.)

[Petke 2017] Justyna Petke, Saemundur Haraldsson, Mark Harman, David White, John Woodward*et al.* *Genetic improvement of software: a comprehensive survey*. IEEE Transactions on Evolutionary Computation, 2017. (Cited on page 41.)

[PezzÃ¨ 2013] Mauro PezzÃ¨, Konstantin Rubinov and Jochen Wuttke. *Generating Effective Integration Test Cases from Unit Ones*. In Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, ICST '13, pages 11–20, Washington, DC, USA, 2013. IEEE Computer Society. (Cited on page 42.)

[Röβler 2012] Jeremias Röβler, Gordon Fraser, Andreas Zeller and Alessandro Orso. *Isolating failure causes through test case generation*. In Proceedings of the 2012 International Symposium on Software Testing and Analysis, pages 309–319. ACM, 2012. (Cited on page 42.)

[Roche 2013] James Roche. *Adopting DevOps Practices in Quality Assurance*. Commun. ACM, vol. 56, 2013. (Cited on page 17.)

[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 49.)

[Spieker 2017] Helge Spieker, Arnaud Gotlieb, Dusica Marijan and Morten Mossige. *Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration*. In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2017, pages 12–22, New York, NY, USA, 2017. ACM. (Cited on page 77.)

[Taneja 2008] K. Taneja and Tao Xie. *DiffGen: Automated Regression Unit-Test Generation*. In Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE '08, pages 407–410, Washington, DC, USA, 2008. IEEE Computer Society. (Cited on page 76.)

[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 8, 18, 19, 47, 51 and 53.)

[Urli 2018] Simon Urli, Zhongxing Yu, Lionel Seinturier and Martin Monperrus. *How to Design a Program Repair Bot? Insights from the Repairnator Project*. In ICSE 2018 - 40th International Conference on Software Engineering, Track Software Engineering in Practice (SEIP), pages 1–10, 2018. (Cited on page 77.)

[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 page 55.)

[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 60.)

[Waller 2015] Jan Waller, Nils C. Ehmke and Wilhelm Hasselbring. *Including Performance Benchmarks into Continuous Integration to Enable DevOps*. SIGSOFT Softw. Eng. Notes, vol. 40, no. 2, pages 1–4, April 2015. (Cited on page 77.)

[Wilkerson 2010] Josh L Wilkerson and Daniel Tauritz. *Coevolutionary automated software correction*. In Proceedings of the 12th annual conference on Genetic and evolutionary computation, pages 1391–1392. ACM, 2010. (Cited on page 41.)

[Xie 2006a] Tao Xie. *Augmenting Automatically Generated Unit-test Suites with Regression Oracle Checking*. In Proceedings of the 20th European Conference on Object-Oriented Programming, pages 380–403, 2006. (Cited on pages 18, 40 and 42.)

[Xie 2006b] 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 8, 10, 19, 47, 51 and 52.)

[Xuan 2014] Jifeng Xuan and Martin Monperrus. *Test Case Purification for Improving Fault Localization*. In Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pages 52–63, New York, NY, USA, 2014. ACM. (Cited on page 18.)

[Xuan 2015] Jifeng Xuan, Xiaoyuan Xie and Martin Monperrus. *Crash Reproduction via Test Case Mutation: Let Existing Test Cases Help*. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pages 910–913, New York, NY, USA, 2015. ACM. (Cited on pages 18 and 43.)

[Yang 2014] Guowei Yang, Sarfraz Khurshid, Suzette Person and Neha Rungta. *Property Differencing for Incremental Checking*. In Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pages 1059–1070, New York, NY, USA, 2014. ACM. (Cited on page 74.)

[Yoo 2012] S. Yoo and M. Harman. *Test Data Regeneration: Generating New Test Data from Existing Test Data*. Softw. Test. Verif. Reliab., vol. 22, no. 3, pages 171–201, May 2012. (Cited on pages 18, 42 and 76.)

[Yu 2013] Zhongxing Yu, Chenggang Bai and Kai-Yuan Cai. *Mutation-oriented Test Data Augmentation for GUI Software Fault Localization*. Inf. Softw. Technol., vol. 55, no. 12, pages 2076–2098, December 2013. (Cited on page 43.)

[Zampetti 2017] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora and M. Di Penta. *How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines*. In 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pages 334–344, May 2017. (Cited on page 77.)

[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 47, 50 and 51.)

[Zhang 2016] Jie Zhang, Yiling Lou, Lingming Zhang, Dan Hao, Lu Zhang and Hong Mei. *Isomorphic Regression Testing: Executing Uncovered Branches Without Test Augmentation*. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, pages 883–894, New York, NY, USA, 2016. ACM. (Cited on page 41.)