Empirical Software Engineering (2020) 25:2379–2415
https://doi.org/10.1007/s10664-019-09794-7

An approach and benchmark to detect behavioral changes of commits in continuous integration

Benjamin Danglot[1] · **Martin Monperrus**[2] · **Walter Rudametkin**[3] · **Benoit Baudry**[2]

Published online: 5 March 2020

Abstract

When a developer pushes a change to an application's codebase, a good practice is to have a test case specifying this behavioral change. Thanks to continuous integration (CI), the test is run on subsequent commits to check that they do no introduce a regression for that behavior. In this paper, we propose an approach that detects behavioral changes in commits. As input, it takes a program, its test suite, and a commit. Its output is a set of test methods that capture the behavioral difference between the pre-commit and post-commit versions of the program. 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 the input space. We evaluate our approach on a curated set of 60 commits from 6 open source Java projects. To our knowledge, this is the first ever curated dataset of real-world behavioral changes. Our evaluation shows that DCI is able to generate test methods that detect behavioral changes. Our approach is fully automated and can be integrated into current development processes. The main limitations are that it targets unit tests and works on a relatively small fraction of commits. More specifically, DCI works on commits that have a unit test that already executes the modified code. In practice, from our benchmark projects, we found 15.29% of commits to meet the conditions required by DCI.

Keywords Continuous Integration · Test amplification · Behavioral change detection

Communicated by: Tao Yue

✉ Benjamin Danglot
bdanglot@gmail.com

Martin Monperrus
martin.monperrus@csc.kth.se

Walter Rudametkin
walter.rudametkin@inria.fr

Benoit Baudry
benoit.baudry@kth.se

[1] INRIA, Lille-Nord Europe, 40 Avenue Halley, Villeneuve d'Ascq, 59650, France

[2] KTH Royal Institute of Technology, Brinellvägen 8, 114 28 Stockholm, Sweden

[3] Université de Lille, 42 rue Paul Duez, 59000 Lille, France

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 and Foemmel 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 et al. 2016; Duvall et al. 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 2006) 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 and Elbaum 2012). We evaluate our approach on a benchmark of 60 commits selected from 6 open source Java projects, constructed with a novel and systematic methodology. We analyzed 1576 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 60 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. This corresponds to 15.3% of commits in average. While this may appear to be a low proportion of commits, our approach is fully automated and developers can still benefit from its output without any manual intervention.

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].

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

```
@@ -260,7 +260,8 @@ public boolean equals(Object object)
} else {
    if (object instanceof FilterStreamType) {
        result = Objects.equals(getType(), ((FilterStreamType) object).getType()
            )
-       && Objects.equals(getDataFormat(),
-                         ((FilterStreamType) object).getDataFormat());
+       && Objects.equals(getDataFormat(),
+                         ((FilterStreamType) object).getDataFormat())
+       && Objects.equals(getVersion(),
+                         ((FilterStreamType) object).getVersion());
    } else {
        result = false;
    }
```

Listing 1 Commit 7E79F77 on XWiki-Commons that changes the behavior without a test

– A comprehensive evaluation based on 4 research questions that combines quantitative and qualitative analysis with manual assessment.

In Section 2 we motivate the need to have commits include a test case that specifies the behavioral change. In Section 3 we introduce our technical contribution: an approach for commit-based test selection and amplification. Section 4 introduces our benchmark of commits, the evaluation protocol and the results of our experiments on 60 real commits. Section 5 discusses the exact applicability scope of our approach. Section 6 presents the threats validity and actions that have been taken to overcome them. In Section 7, we expose the related work, their evaluation and the differences with our work and eventually we conclude in Section 8.

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.

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 Listing 1, adds a comparison to ensure the equality of the objects returned by `getVersion()`. The developer did not write a test method nor modify an existing one.

In this commit, the intent is to take into account the `version` (from method `getVersion`) in the `equals` method. This change impacts the behavior of all methods that use it, `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.

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

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 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.

2.2 Practibility

We describe a complete scenario to sum up the 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 et al. 2009).

DCI could apply to any kind of test: unit-level or system-level. However, from our experience, unit tests (vs integration tests) are the best target for DCI, for two reasons. First, they have a small scope, which allows 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 execute compared to integration tests.

Since DCI needs to execute the tests 5 times 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.

2.3 Behavioral Change

A *behavioral change* is a source-code modification that triggers a new state for some inputs (Saff and Ernst 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.

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.

2.5 Test Amplification

Test amplification is the idea of improving existing tests with respect to a specific test criterion (Zhang and Elbaum 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.

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 to use it during continuous integration.

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 1 sums up the different phases of the approach:

1) Compute the diff coverage and select the test methods 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.

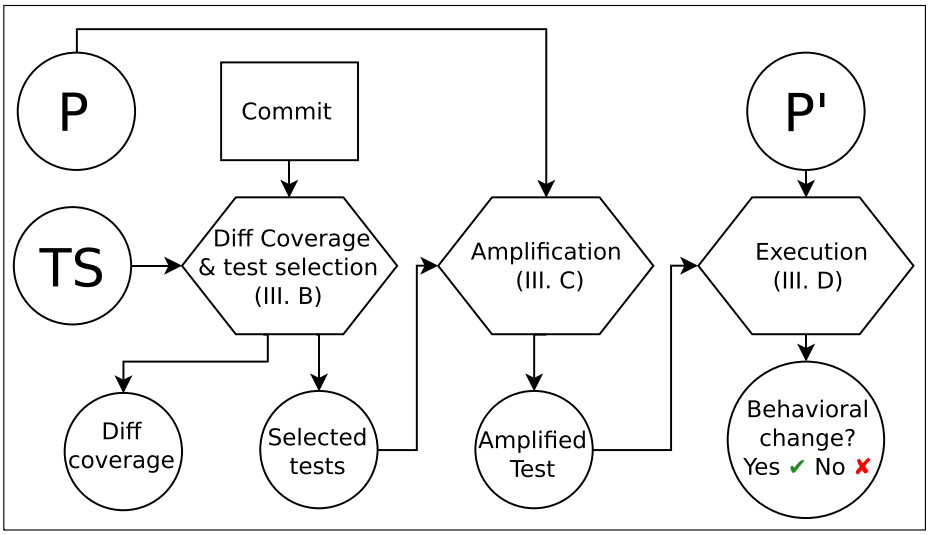

Fig. 1 Overview of our approach to detect behavioral changes in commits

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.

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 and Elbaum 2012). In DCI, we build upon Xie's technique (Xie 2006) and Tonella's evolutionary algorithm (Tonella 2004) to perform test amplification.

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 (2006). 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 1 AAMPL: Assertion amplification algorithm.

Require: Program P
Require: Test Suite TS
Ensure: An Amplified Test Suite ATS
1: $ATS \leftarrow \emptyset$
2: **for** $Test$ in TS **do**
3: $NoAssertTest \leftarrow removeAssertions(Test)$
4: $InstrTest \leftarrow instrument(NoAssertTest)$
5: $execute(InstrTest)$
6: $AmplTest \leftarrow NoAssertTest.clone()$
7: **for** $Observ$ in $InstrTest.observations()$ **do**
8: $Assert \leftarrow generateAssertion(Observ)$
9: $AmplTest \leftarrow AmplTest.add(Assert)$
10: **end for**
11: $ATS.add(select(AmplTest))$
12: $ATS.add(AmplTest)$
13: **end for**
14: **return** ATS

We present AAMPL's pseudo-code in Algorithm 1. 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.

3.3.2 Search-based Amplification

Search-based test amplification consists in running stochastic transformations on test code (Tonella 2004).

For DCI_{SBAMPL}, this process consists in

a) generating a set of original test methods by applying code transformations;
b) executing AAMPL to synthesize new assertions for these test methods for which the input has been modified at the previous step;
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 2 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 for
12: return ATS
```

We present the search-based amplification algorithm in Algorithm 2. 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 a behavioral change);

and Finally, 4) affects $AmplifiedTests$ to $TmpTests$ in order to stack transformations. In our study, we consider the test transformations in Table 1.

[3]by default, $nb = 3$

Table 1 Test transformations considered in our study

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

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

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.

3.5 Implementation

DCI is implemented in Java and is built on top of the OpenClover and Gumtree (Falleri et al. 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 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?

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

4.1 Benchmark

To the best of our knowledge, there is no benchmark of commits in Java with real 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 et al. (2018) and Danglot et al. (2019), which is composed of mature Java projects from GitHub.

Commit selection We take commits in inverse chronological order, from newest to oldest. On September 10 2018, we selected the first 10 commits that match the following criteria:

- The commit modifies Java files (most behavioral changes are source code changes.[5]).
- 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.
- 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**.

Together, these criteria ensure that all selected commits:

1) modify java files,
2) that there is at least 1 test in the pre-commit version of the program that executes the diff and can be used to seed the amplification process
3) 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
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.)

Final benchmark Table 2 shows the main descriptive statistics of the benchmark dataset. The *project* column is the name of the project. The *LOC* column is the number of lines of code computed with *cloc*. The *start date* column is the date of the project's oldest commit. The *end date* column is the date of the project's newest commit. The *#total commit* column is the total number of commits we analyzed.

#Matching commits is the number of commits that match our first two criteria to run DCI but might not provide a test in the post-commit version that fails on the pre-commit version of the program. We could potentially apply DCI on all *#matching commits*, but for this paper, we cannot validate DCI with them because they might not provide a ground-truth

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

Table 2 Considered period for selecting commits

Project	LOC	start date	end date	#total commits	#matching commits	#selected commits
commons-io	59607	9/10/2015	9/29/2018	385	49 / 12.73%	10
commons-lang	77410	11/22/2017	10/9/2018	227	40 / 17.62%	10
gson	49766	6/14/2016	10/9/2018	159	56 / 35.22%	10
jsoup	20088	12/21/2017	10/10/2018	50	42 / 84.00%	10
mustache.java	10289	7/6/2016	04/18/2019	68	28 / 41.18%	10
xwiki-commons	87289	10/31/2017	9/29/2018	687	26 / 3.78%	10
summary	304449	9/10/2015	04/18/2019	avg(262.67)	avg(40.17 / 15.29%)	60

test. The *#selected commits* column shows the number of commits we select for evaluation. It is a subset of *#matching commits* from which we searched for the first 10 commits per project that match all criteria, including a ground-truth test to evaluate DCI.

The bottom row reports a summary of the benchmark dataset with the total number of lines of code, the oldest and the newest commit dates, the average number of commits analyzed, the average number of commits matching all the criteria but the third: there is a test in the post-commit version of the program that detect the behavioral change, and the total number of selected commits. The percentage in parenthesis next to the averages are percentage of averages, *e.g.* $\frac{\#matching}{\#total}$. 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.

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. In addition, we want to have a proper understanding of the impact of randomness as follows. We consider the case of $n = 1$ iteration. For "$n = 1$, we run DCI for each commit for 10 different seeds in addition to the reference run with the default seed, totalling 11 runs.. From those runs, we compute the confidence interval on the number of successes, *i.e.* the number of time DCI generates at least one amplified test method that detects the behavioral change, in order to measure the uncertainty of the result. To do this, we use Python libraries *scipy* and *numpy*, and we consider a confidence level of 95%. Per our open-science approach, the interested reader has access to both the raw data and the script computing the confidence interval.[6]

For **RQ3**, the test selection method is considered effective if the tests selected for amplification 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.

4.3 Results

The overall results are reported in Table 3. This table can be read as follows: the first column is the name of the project; the second column is the shortened commit id; the third column is the commit date; the fourth column column is the total number of test methods executed when building that version of the project; the fifth and sixth columns are respectively the number of tests modified or added by the commit, and the size of the diff in terms of line

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

Table 3 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	8	2.4m	1608	6.5h
	average						6.60	0.80	14.5s	160.80	38.8m
commons-lang	f56931c	7/2/18	4105	1	30 / 4	25.0	42	0	2.4m	0	8.5m
	87937b2	5/22/18	4101	1	114 / 0	77.78	16	0	35.0s	0	18.1m
	09ef69c	5/18/18	4100	1	10 / 1	100.0	4	0	16.0s	0	98.8m
	3fadfdd	5/10/18	4089	1	7 / 1	100.0	9	0	17.0s	✓(4)	17.2m
	e7d16c2	5/9/18	4088	1	13 / 1	33.33	7	0	16.0s	✓(2)	15.1m
	50ce8c4	3/8/18	4084	4	40 / 1	90.91	2	✓(1)	28.0s	✓(135)	2.0m
	2e9f3a8	2/11/18	4084	2	79 / 4	30.0	47	0	79.0s	0	66.5m
	c8e61af	2/10/18	4082	1	8 / 1	100.0	10	0	17.0s	0	16.0s
	d8ec011	11/12/17	4074	1	11 / 1	100.0	5	0	31.0s	0	2.3m
	7d061e3	11/22/17	4073	1	16 / 1	100.0	8	0	17.0s	0	11.4m
	total						150	1	6.7m	141	4.0h
	average						15.00	0.10	40.5s	14.10	24.0m

Table 3 (continued)

	id	date	#Test	#Modified tests	+ / -	Cov	#Selected tests	#AAMPL tests	Time	#SBAMPL tests	Time
gson	b1fb9ca	9/22/17	1035	1	23 / 0	50.0	166	0	4.2m	0	92.5m
	7a9fd59	9/18/17	1033	2	21 / 2	83.33	14	0	15.0s	✓(108)	2.1m
	03a72e7	8/1/17	1031	2	43 / 11	68.75	371	0	7.7m	0	3.2h
	74e3711	6/20/17	1029	1	68 / 5	8.0	1	0	4.0s	0	16.0s
	ada597e	5/31/17	1029	2	28 / 3	100.0	5	0	8.0s	0	8.7m
	a300148	5/31/17	1027	7	103 / 2	18.18	665	0	9.2m	✓(6)	4.9h
	9a24219	4/19/17	1019	1	13 / 1	100.0	36	0	2.2m	0	48.9m
	9e6f2ba	2/16/17	1018	2	56 / 2	50.0	9	0	32.0s	✓(2)	8.5m
	44cad04	11/26/16	1015	1	6 / 0	100.0	2	0	15.0s	✓(37)	40.0s
	b2c00a3	6/14/16	1012	4	242 / 29	60.71	383	0	7.9m	0	3.6h
	total						1652	0	32.4m	153	14.4h
	average						165.20	0.00	3.2m	15.30	86.5m
jsoup	426ffe7	5/11/18	668	4	27 / 46	64.71	27	✓(2)	42.0s	✓(198)	33.6m
	a810d2e	4/29/18	666	1	27 / 1	80.0	5	0	10.0s	0	26.6m
	6be19a6	4/29/18	664	1	23 / 1	50.0	50	0	69.0s	0	67.7m
	e38dfd4	4/28/18	659	1	66 / 15	90.0	18	0	35.0s	0	12.5m
	e9feec9	4/15/18	654	1	15 / 3	100.0	4	0	9.0s	0	95.0s
	0f7e0cc	4/14/18	653	2	56 / 15	84.62	330	0	6.5m	✓(36)	11.8h
	2c4e79b	4/14/18	650	2	82 / 2	50.0	44	0	67.0s	0	4.7h
	e5210d1	12/22/17	647	1	3 / 3	100.0	14	0	9.0s	0	4.9m
	df272b7	12/22/17	647	2	17 / 1	100.0	13	0	9.0s	0	4.6m
	3676b13	12/21/17	648	6	104 / 12	38.46	239	0	6.2m	✓(52)	6.8h
	total						744	2	16.8m	286	25.8h
	average						74.40	0.20	101.0s	28.60	2.6h

Table 3 (continued)

	id	date	#Test	#Modified tests	+ / -	Cov	#Selected tests	#AAMPL tests	Time	#SBAMPL tests	Time
mustache.java	a1197f7	1/25/18	228	1	43 / 57	77.78	131	0	11.8m	✓(204)	10.1h
	8877027	11/19/17	227	1	22 / 2	33.33	47	0	7.3m	0	100.2m
	d8936b4	2/1/17	219	2	46 / 6	60.0	168	0	12.7m	0	84.2m
	88718bc	1/25/17	216	2	29 / 1	100.0	1	✓(1)	7.0s	✓(149)	3.7m
	339161f	9/23/16	214	2	32 / 10	77.78	123	0	8.6m	✓(1312)	5.8h
	774ae7a	8/10/16	214	2	17 / 2	100.0	11	0	66.0s	✓(124)	6.8m
	94847cc	7/29/16	214	2	17 / 2	100.0	95	0	11.5m	✓(2509)	21.4h
	eca08ca	7/14/16	212	4	47 / 10	80.0	18	0	87.0s	0	41.8m
	6d7225c	7/7/16	212	2	42 / 4	80.0	18	0	87.0s	0	40.1m
	8ac71b7	7/6/16	210	10	167 / 31	40.0	20	0	2.1m	✓(124)	5.6m
	total						632	1	58.1m	4422	42.0h
	average						63.20	0.10	5.8m	442.20	4.2h
xwiki-commons	ffc3997	7/27/18	1081	0	125 / 18	21.05	1	0	29.0s	0	18.0s
	ced2635	8/13/18	1081	1	21 / 14	60.0	5	0	93.0s	0	2.5h
	10841b1	8/1/18	1061	1	107 / 19	30.0	51	0	5.7m	0	3.4h
	848c984	7/6/18	1074	1	154 / 111	17.65	1	0	28.0s	0	18.0s
	adfefec	6/27/18	1073	1	17 / 14	40.0	22	✓(1)	76.0s	✓(3)	14.9m
	d3101ae	1/18/18	1062	2	71 / 9	20.0	4	✓(1)	72.0s	✓(31)	41.4m
	a0e8b77	1/18/18	1062	2	51 / 8	42.86	4	✓(1)	72.0s	✓(60)	42.1m
	78ff099	12/19/17	1061	1	16 / 0	33.33	2	0	68.0s	✓(4)	6.6m
	1b79714	11/13/17	1060	1	20 / 5	60.0	22	0	78.0s	0	17.9m
	6dc9059	10/31/17	1060	1	4 / 14	88.89	22	0	79.0s	0	20.5m
	total						134	3	15.7m	98	8.2h
	average						13.40	0.30	94.3s	9.80	49.5m
	total						3378	9(15)	2.2h	25(6708)	100.9h

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

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

In this section, we describe the characteristics of commits introducing behavioral changes in the context of continuous integration The first five columns in Table 3 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 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 Fig. 2: 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 60 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.

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 3. 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 detects the change.

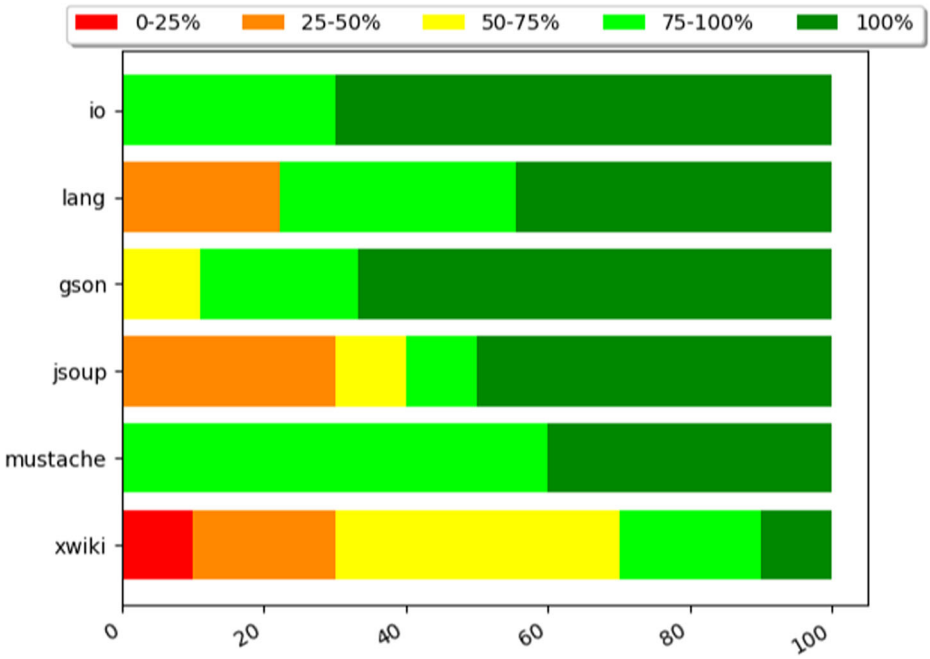

Fig. 2 Distribution of diff coverage per project of our benchmark

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_{SBAMPL} 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 42% of the commits in our benchmark (25 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 and Miller 1995). Consequently, the fact that DCI can generate test cases that detect behavioral changes, is considered as an achievement. The commits for which DCI fails to detect the change can be considered as a target for future research on this topic.

```
1 @@ -2619,7 +2619,7 @@ protected void appendFieldStart(final StringBuffer buffer,
      final String fieldNam
2
3 -         super.appendFieldStart(buffer, FIELD_NAME_QUOTE + fieldName
4 +         super.appendFieldStart(buffer, FIELD_NAME_QUOTE +
5 +             StringEscapeUtils.escapeJson(fieldName) + FIELD_NAME_QUOTE);
6     }
```

Listing 2 Diff of commit 3FADFDD from commons-lang

Now, we manually analyze a successful case where DCI detects the behavioral change. We select commit 3FADFDD[7] from commons-lang, which is succinct enough to be discussed in the paper. The diff is shown in Listing 2.

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

DCI starts the amplification from the `testNestingPerson` test method defined in `JsonToStringStyleTest`, showed in Listing 3.

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

We show in Listing 4 the resulting amplified test method. In this generated test, DCI_{SBAMPL} applies 2 input transformations: 1 duplication of method call and 1 character replacement in an existing String literal. The latter transformation is the key transformation: DCI replaced an 'a' inside "name" by '/' resulting in "n/me" where "/" is a special character that must be escaped (Line 8). 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:

`{\"n/me\":\"Jane Doe\",\"age\":25,\"smoker\":true}`

while in the post-commit version it is

`{\"n\/me\":\"Jane Doe\",\"age\":25,\"smoker\":true}` (Line 9).

> Answer to **RQ1**: Overall, DCI is capable of detecting the behavioral changes for 25/60 commits. DCI_{SBAMPL} finds behavioral changes in 25/60 commits, while DCI_{AAMPL} finds some in 9/60 commits. Since DCI_{SBAMPL} also uses AAMPL to generate assertions, all DCI_{AAMPL}'s commits are contained in DCI_{SBAMPL}'s. The search-based algorithm of input exploration finds many more behavioral changes, at the cost of execution time.

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

The results are reported in Table 4.

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 the number of amplified tests that detect a behavioral change and a - denotes that DCI did not succeed in generating a test that detects a change. The last row reports the total over the 6 projects. For the third, fifth and the seventh columns of the last row, the first number is the number of successes, *i.e.* the number of times that DCI produced at least

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

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

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

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

one amplified test method that detect the behavioral change, for respectively$iteration = 1$, $iteration = 2$ and $iteration = 3$. The numbers in parentheses are the total number of amplified test methods that DCI produces with each number of iteration.

Overall, DCI$_{SBAMPL}$ generates amplified tests that detect 23, 24, and 25 out of 60 behavioral changes for respectively $iteration = 1$, $iteration = 2$ and $iteration = 3$. The more iteration DCI$_{SBAMPL}$ 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 2 more behavioral changes than when it is used with $iteration = 1$ and 1 then when it is used with $iteration = 2$.

On 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 $iteration = 3$.

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

Listing 4 Test generated by DCI that detects the behavioral change of 3FADFDD from commons-lang

Table 4 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	477	2.0h	1058	4.0h	1608	6.5h
	average	47.70	12.3m	105.80	24.0m	160.80	38.8m
commons-lang	f56931c	0	0.0s	0	3.7m	0	8.5m
	87937b2	0	3.5m	0	10.5m	0	18.1m
	09ef69c	0	97.0s	0	21.0m	0	98.8m
	3fadfdd	✓(1)	2.0m	✓(1)	9.3m	✓(4)	17.2m
	e7d16c2	✓(3)	111.0s	✓(2)	8.4m	✓(2)	15.1m
	50ce8c4	✓(61)	38.0s	✓(97)	78.0s	✓(135)	2.0m
	2e9f3a8	0	11.4m	0	35.0m	0	66.5m
	c8e61af	0	16.0s	0	16.0s	0	16.0s
	d8ec011	0	36.0s	0	68.0s	0	2.3m
	7d061e3	0	79.0s	0	5.8m	0	11.4m
	total	65	23.3m	100	96.4m	141	4.0h
	average	6.50	2.3m	10.00	9.6m	14.10	24.0m
gson	b1fb9ca	0	14.6m	0	51.0m	0	92.5m
	7a9fd59	✓(7)	33.0s	✓(48)	73.0s	✓(108)	2.1m
	03a72e7	0	30.2m	0	102.3m	0	3.2h
	74e3711	0	6.0s	0	11.0s	0	16.0s
	ada597e	0	61.0s	0	4.9m	0	8.7m
	a300148	0	45.2m	✓(4)	2.6h	✓(6)	4.9h
	9a24219	0	10.8m	0	28.4m	0	48.9m
	9e6f2ba	0	79.0s	0	4.5m	✓(2)	8.5m
	44cad04	✓(4)	21.0s	✓(21)	30.0s	✓(37)	40.0s
	b2c00a3	0	31.5m	0	111.8m	0	3.6h
	total	11	2.3h	73	7.7h	153	14.4h
	average	1.10	13.6m	7.30	46.0m	15.30	86.5m

Table 4 (continued)

	id	$it = 1$	Time	$it = 2$	Time	$it = 3$	Time
jsoup	426ffe7	✓(126)	5.4m	✓(172)	19.2m	✓(198)	33.6m
	a810d2e	0	90.0s	0	13.9m	0	26.6m
	6be19a6	0	8.1m	0	39.7m	0	67.7m
	e38dfd4	0	117.0s	0	6.3m	0	12.5m
	e9feec9	0	20.0s	0	50.0s	0	95.0s
	0f7e0cc	✓(1)	2.4h	✓(7)	6.8h	✓(36)	11.8h
	2c4e79b	0	7.1m	0	34.1m	0	4.7h
	e5210d1	0	45.0s	0	2.3m	0	4.9m
	df272b7	0	43.0s	0	2.2m	0	4.6m
	3676b13	✓(6)	21.4m	✓(35)	2.9h	✓(52)	6.8h
	total	133	3.2h	214	11.6h	286	25.8h
	average	13.30	19.4m	21.40	69.8m	28.60	2.6h
mustache.java	a1197f7	✓(28)	5.9h	✓(124)	8.4h	✓(204)	10.1h
	8877027	0	30.5m	0	58.4m	0	100.2m
	d8936b4	0	3.2m	0	4.8m	0	84.2m
	88718bc	✓(13)	78.0s	✓(85)	2.5m	✓(149)	3.7m
	339161f	✓(143)	115.9m	✓(699)	4.1h	✓(1312)	5.8h
	774ae7a	✓(18)	2.7m	✓(65)	4.7m	✓(124)	6.8m
	94847cc	✓(122)	5.3h	✓(580)	10.4h	✓(2509)	21.4h
	eca08ca	0	8.1m	0	24.3m	0	41.8m
	6d7225c	0	7.9m	0	26.8m	0	40.1m
	8ac71b7	✓(2)	2.7m	✓(48)	3.8m	✓(124)	5.6m
	total	326	14.0h	1601	25.0h	4422	42.0h
	average	32.60	84.3m	160.10	2.5h	442.20	4.2h
xwiki-commons	ffc3997	0	19.0s	0	18.0s	0	18.0s
	ced2635	0	8.0m	0	31.8m	0	2.5h
	10841b1	0	56.2m	0	2.9h	0	3.4h
	848c984	0	18.0s	0	17.0s	0	18.0s
	adfefec	✓(22)	3.5m	✓(57)	9.9m	✓(3)	14.9m
	d3101ae	✓(9)	11.6m	✓(12)	28.2m	✓(31)	41.4m
	a0e8b77	✓(10)	12.0m	✓(17)	28.2m	✓(60)	42.1m
	78ff099	✓(4)	2.6m	✓(4)	4.6m	✓(4)	6.6m
	1b79714	0	4.0m	0	10.7m	0	17.9m
	6dc9059	0	4.0m	0	10.8m	0	20.5m
	total	45	102.8m	90	4.9h	98	8.2h
	average	4.50	10.3m	9.00	29.7m	9.80	49.5m
	total	23(1057)	23.7h	24(3136)	54.9h	25(6708)	100.9h

Table 5 Number of successes, *i.e.* DCI produced at least one amplified test method that detects the behavioral changes, for 10 different seeds

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

Impact of the randomness The number of amplified test methods obtained by the different seeds are reported in Table 5.

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

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

Answer to **RQ2**: DCI_{SBAMPL} detects 23, 24, and 25 behavioral changes out of 60 commits for respectively $iteration = 1$, $iteration = 2$ and $iteration = 3$. The number of iterations performed by DCI_{SBAMPL} impacts the number of behavioral changes detected, the number of amplified test methods obtained and the execution time.

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 Section 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[8]: 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 of 3 (`testCopyURLToFileWithTimeout` and `testCopyURLToFile`) have an intention related to the changed file. The selection is thus considered relevant.

COMMONS-LANG#F56931C[9]: our test selection returns 39 test methods from 5 test classes: `FastDateFormat_ParserTest`, `FastDateParserTest`, `DateUtilsTest`, `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

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

to dates. While this is meaningful, a set of 39 methods is not a focused selection. It is considered as an half-success.

GSON#9E6F2BA[10]: 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[11], 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, manually written 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[12], 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[13] 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 the 6 manually analyzed cases, the tests selected to be amplified are semantically related to the modified application code. In the 2 remaining cases, DCI selects tests whose intention is semantically relevant to the change, but also tests that are not. DCI's test selection provides developers with important and targeted context to better understand the behavioral change at hand.

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.[14] If they capture the same behavioral change, it means they have the same intention and we consider the amplification a success.

[10] https://github.com/google/gson/commit/9e6f2ba
[11] https://github.com/jhy/jsoup/commit/e9feec9
[12] https://github.com/spullara/mustache.java/commit/88718bc
[13] https://github.com/xwiki/xwiki-commons/commit/848c984
[14] For a side-by-side comparison, see https://danglotb.github.io/resources/dci/index.html

```
1  @Test(timeout = 10000)
2  public void readMulti_literalMutationNumber3() {
3      BoundedReader mr = new BoundedReader(sr, 0);
4      char[] cbuf = new char[4];
5      for (int i = 0; i < (cbuf.length); i++) {
6          cbuf[i] = 'X';
7      }
8      final int read = mr.read(cbuf, 0, 4);
9      Assert.assertEquals(0, ((int) (read)));
10 }
```

Listing 5 Test generated by DCI_{SBAMPL} that detects the behavioral change introduced by commit 81210EB in commons-io

COMMONS-IO#81210EB[15]: This commit modifies the behavior of the `read()` method in `BoundedReader`. Listing 5 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 (Line 3) 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.

Displayed Fx fig

Now, let us look at the human test contained in the commit, shown in Listing 6. It captures the behavioral change with the timeout (the test timeouts on the pre-commit version and goes fast enough on the post-commit version). Furthermore, it only indirectly calls the changed method through a call to `readLine`.

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

COMMONS-LANG#E7D16C2[16]: this commit escapes special characters before adding them to a `StringBuffer`. Listing 7 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.* '\—'. In the post-commit version, the DCI test fails since the code now escapes the special character \.

Let's have a look ar the human test method shown in Listing 8. 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[17]: This commit allows Gson to deserialize a number represented as a string. Listing 9 shows the relevant part of the test generated by DCI_{SBAMPL}, based on `testNumberDeserialization` of `PrimitiveTest` as a seed. First, we see 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,

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

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

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

```
1 @Test(timeout = 5000)
2 public void testReadBytesEOF() {
3     BoundedReader mr = new BoundedReader( sr, 3 );
4     BufferedReader br = new BufferedReader( mr );
5     br.readLine();
6     br.readLine();
7 }
```

Listing 6 Developer test for commit 81210EB of commons-io

line 4 throws a `JsonSyntaxException`. On the post-commit version, line 5 throws a `NumberFormatException`. In other words, the behavioral change is detected by a different exception (different type and not thrown at the same line).[18].

We compare it against the developer-written ground-truth method, shown in Listing 10. 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[19]: 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. Listing 11 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.

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 (Listing 12).

MUSTACHE.JAVA#774AE7A[20]: This commit fixes an issue with the usage of a dot in a relative path on Window in the method `getReader` of class `ClasspathResolver`. The test method `getReaderNullRootDoesNotFindFileWithAbsolutePath` has been used as seed by DCI. It modifies the existing string literal with another string used somewhere else in the test class and generates 3 new assertions (Listing 13). The behavioral change is detected thanks to the modified strings: it produces the right test case containing a space.

The developer proposed two tests that verify that the object reader is not null when getting it with dots in the path. There are shown in Listing 14. These tests invoke the method `getReader` which is the modified method in the commit.

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

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

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

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

Listing 7 Test generated by DCI$_{SBAMPL}$ that detects the behavioral change of E7D16C2 in commons-lang

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

Listing 8 Developer test for E7D16C2 of commons-lang

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

Listing 9 Test generated by DCI that detects the behavioral change of commit 44CAD04 in Gson

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

Listing 10 Provided test by the developer for 44CAD04 of Gson

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

Listing 11 Test generated by DCI$_{SBAMPL}$ that detects the behavioral change of 3676B13 of Jsoup

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

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

Listing 12 Provided test by the developer for 3676B13 of Jsoup

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

Listing 13 Test generated by DCI$_{SBAMPL}$ that detects the behavioral change of 774AE7A of Mustache.java

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

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

Listing 14 Developer test for 774AE7A of Mustache.java

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.

XWIKI-COMMONS#D3101AE[21]: This commit fixes a bug in the `merge` method of class `DefaultDiffManager`. Listing 15 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 such a behavioral change in the diff, with the addition of a call to "logConflict" in the newly handled case.

The developer's test is shown in Listing 16. 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.

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

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

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

Listing 15 Test generated by DCI$_{AAMPL}$ that detects the behavioral change of D3101AE of XWiki

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

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

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

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

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

Listing 16 Developer test for D3101AE of XWiki

5 Discussion About the Scope of DCI

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

Focused applicability From our benchmark, we see that DCI is applicable to a limited proportion of commits: on average 15.29% of the commits analyzed. This low proportion is the first limit of DCI usage. However, once DCI is setup, it is fully automated, there is no manual overhead. Even if DCI is not used at each commit, it costs little more.

Adoption Our evaluation showed that DCI is able to obtain amplified test methods that detect behavioral changes. But, it does not provide any evidence on the fact that developers would exploit such test methods. However, from our past evaluation (Danglot et al. 2019), we know that software developers value amplified test methods. This provides strong evidence of the potential adoption of DCI.

Performance From our experiments, we see that the time to complete the amplification is the main limitation of DCI. For example DCI took almost 5 hours on JSOUP#2C4E79B, with no result. For the sake of our experimentation, we choose to use a pre-defined number of iterations 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 seeds. For example, see the row of `commons-lang#c8e61af` in Table 4, where one can observe that whatever the number of iterations, DCI takes the same time to complete the amplification. The reason is that the seed tests are only composed of assertions statements. Such tests are bad seeds for DCI, and they prevent any good input amplification. Also, DCI requires to have at least one test method that executes the code changes. If the project is poorly tested and does not have any test method that execute the code changes, DCI cannot be applied.

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

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, there is a potential flakiness to 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 each amplified test 3 times in order to check weather or not there are stable. If the outcome of at least one execution is different than the others, we discard the amplified test.

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

7 Related Work

7.1 Commit-based Test Generation

Person et al. (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 (2013) present Katch, a system that aims at covering the code included in a patch. This approach first determine[17.66 ; 20.34s 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 et al. 2016; Kuchta et al. 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 et al. 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 and Roychoudhury 2014) data set that contains documented regression bugs of the GNU Coreutils program suite.

Noller et al. (2018) aim at detecting regression bugs. They apply shadow symbolic execution, originally from Palikevera (Person et al. 2008; Palikareva et al. 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 et al. 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. [17.66 ; 20.34 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. (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. (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. (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. (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.

7.2 Behavioral Change Detection

Evans and Savoia (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.

Jin et al. (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 and Xie (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 modify 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.

Madeiral et al. (2019) built a benchmark of bugs for evaluating automatic program repair tools. This benchmark has been built using behavioral change detection such as we d do in this paper. However, this benchmark includes a different kind of behavioral change: bug fixes. Also, they have different criteria to select the commits than ours, and their procedure is similar in different ways. Their approach used continuous integration to build automatically and enrich their benchmark, and it would be fruitful to automate our process as well.

7.3 Test Amplification

Yoo and Harman (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 et al. 2005), search-based test generation (Fraser and Arcuri 2012), or random test generation (Groce et al. 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

7.4 Continuous Integration

Hilton et al. (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. (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. (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. (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.

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 analyzed 1576 commits from 6 projects. On average, our approach is applicable to 15.29% of commits per-project. We built a curated set of 60 commits coming from real-world, large open-source Java projects to evaluate our technique. We show that our approach is able to detect the behavioral differences of 25 of the 60 commits.

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 et al. 2018).

References

Anand S, Pasareanu CS, Visser W (2007) Jpf-se: A symbolic execution extension to java pathfinder 03

Böhme M, Roychoudhury A (2014) Corebench: Studying complexity of regression errors. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. ACM, pp 105–115

Cadar C, Dunbar D, Engler D (2008) 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. USENIX Association, Berkeley, pp 209–224

Campos J, Arcuri A, Fraser G, Abreu R (2014) 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. ACM, pp 55–66

Danglot B, Vera-Pérez OL, Baudry B, Monperrus M (2019) Automatic test improvement with dspot: a study with ten mature open-source projects. Empirical Software Engineering

Daniel B, Jagannath V, Dig D, Marinov D (2009) Reassert: Suggesting repairs for broken unit tests. In: 2009 IEEE/ACM International conference on automated software engineering, pp 433–444

Duvall PM, Matyas S, Glover A (2007) Continuous integration: improving software quality and reducing risk. Pearson Education

Evans RB, Savoia A (2007) 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. ACM, pp 549–552

Falleri J-R, Morandat F, Blanc X, Martinez M, Monperrus M (2014) Fine-grained and Accurate Source Code Differencing. In: Proceedings of the International Conference on Automated Software Engineering, pp 313–324

Fowler M, Foemmel M (2006) Continuous integration. Thought-Works https://www.thoughtworks.com/continuous-integration, pp 122:14

Fraser G, Arcuri A (2012) The seed is strong: Seeding strategies in search-based software testing. In: 2012 IEEE fifth international conference on Software testing, verification and validation (ICST). IEEE, pp 121–130

Godefroid P, Klarlund N, Sen K (2005) Dart: directed automated random testing. In: ACM Sigplan notices. ACM, vol 40, pp 213–223

Groce A, Holzmann G, Joshi R (2007) Randomized differential testing as a prelude to formal verification. In: Proceedings of the 29th international conference on Software Engineering. IEEE Computer Society, pp 621–631

Hilton M, Tunnell T, Huang K, Marinov D, Dig D (2016) 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 .ACM, New York, pp 426–437

Hilton M, Bell J, Marinov D (2018) A large-scale study of test coverage evolution. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018. ACM, New York, pp 53–63

Jin W, Orso A, Xie T (2010) Automated behavioral regression testing. In: 2010 Third international conference on software testing, verification and validation, pp 137–146

Kuchta T, Palikareva H, Cadar C (2018) Shadow symbolic execution for testing software patches. ACM Trans Softw Eng Methodol 27(3):10:1–10:32

Lahiri S, McMillan K, Hawblitzel C (2013) Differential assertion checking. Technical report

Madeiral F, Urli S, Maia M, Monperrus M (2019) Bears An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In: Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '19)

Marinescu PD, Cadar C (2013) KATCH: high-coverage testing of software patches. ACM Press, pp 235

Menarini M, Yan Y, Griswold WG (2017) Semantics-assisted code review: an efficient tool chain and a user study. In: 2017 32Nd IEEE/ACM international conference on automated software engineering (ASE), pp 554–565

Noller Y, Nguyen HL, Tang M, Kehrer T (2018) Shadow symbolic execution with java pathfinder. SIGSOFT Softw. Eng. Notes 42(4):1–5
Palikareva H, Kuchta T, Cadar C (2016) Shadow of a doubt: testing for divergences between software versions. In: Proceedings of the 38th International Conference on Software Engineering. ACM, pp 1181–1192
Person S, Dwyer MB, Elbaum S, Păsăreanu CS (2008) Differential symbolic execution. In: sProceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '08/FSE-16. ACM, New York, pp 226–237, NY
Saff D, Ernst MD (2004) An experimental evaluation of continuous testing during development. In: ACM SIGSOFT Software engineering notes. ACM, vol 29, pp 76–85
Spieker H, Gotlieb A, Marijan D, Mossige M (2017) 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. ACM, New York, pp 12–22
Taneja K, Xie T (2008) Diffgen: Automated regression unit-test generation. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE '08. IEEE Computer Society, Washington, pp 407–410
Tonella P (2004) Evolutionary testing of classes. In: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '04. ACM, New York, pp 119–128
Urli S, Yu Z, Seinturier L, Monperrus M (2018) 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), pp 1–10
Vera-Pérez OL, Danglot B, Monperrus M, Baudry B (2018) A comprehensive study of pseudo-tested methods. Empirical Software Engineering
Voas JM, Miller KW (1995) Software testability: the new verification. IEEE Softw 12(3):17–28
Waller J, Ehmke NC, Hasselbring W (2015) Including performance benchmarks into continuous integration to enable devops. SIGSOFT Softw Eng Notes 40(2):1–4
Xie T (2006) Augmenting automatically generated unit-test suites with regression oracle checking. In: Thomas D (ed) ECOOP 2006 – Object-Oriented Programming. Springer, Berlin, pp 380–403
Yang G, Khurshid S, Person S, Rungta N (2014) Property differencing for incremental checking. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014. ACM, New York, pp 1059–1070
Yoo S, Harman M (2012) Test data regeneration: Generating new test data from existing test data. Softw Test Verif Reliab 22(3):171–201
Zampetti F, Scalabrino S, Oliveto R, Canfora G, Penta MD (2017) 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), pp 334–344
Zhang P, Elbaum S (2012) Amplifying tests to validate exception handling code. In: Proc. of int. Conf. on software engineering (ICSE). IEEE Press, pp 595–605

Benjamin Danglot is a researcher in the following fields: test suite amplification in the DevOps context and chaos engineering. His work took part in the Horizon 2020 European project called STAMP: Software Testing AMPlification.

Walter Rudametkin is an associate professor at the University of Lille and part of the Spirals team, a joint team between the CRIStAL laboratory and Inria. He received his Ph.D. in 2013 from the University of Grenoble, focused on dynamic updates on large software software systems. His work currently focuses on applying software engineering to cross-cutting concerns, such as privacy and security.

Martin Monperrus is Professor of Software Technology at KTH Royal Institute of Technology, Sweden. In 2011-2017, he was associate professor at the University of Lille, France and adjunct researcher at Inria. He received a Ph.D. from the University of Rennes, and a Master's degree from the Compiègne University of Technology. His research lies in the field of software engineering with a current focus on automatic program repair, self-healing software and chaos engineering.

Benoit Baudry is a Professor in Software Technology at the Royal Institute of Technology (KTH) in Stockholm, Sweden. He received his PhD in 2003 from the University of Rennes and was a research scientist at INRIA from 2004 to 2017. His research is in the area of software testing, code analysis and automatic diversification. He has led the largest research group in software engineering at INRIA, as well as collaborative projects funded by the European Union, and software companies