



STAMP

Deliverable D1.4

Consolidated tool for the unit test amplification, selection and execution



Project Number	:	731529
Project Title	:	STAMP
Deliverable Type	:	Report

Deliverable Number	:	D1.4
Title of Deliverable	:	Consolidated tool for the unit test amplification, selection and execution
Dissemination Level	:	Public
Version	:	1.00
Latest version	:	https://github.com/STAMP-project/docs-forum/blob/master/docs/d14_consolidated_unit_test_amplification_tool.pdf
Contractual Delivery Date	:	M20 July, 31 2018
Contributing WPs	:	WP 1
Editor(s)	:	Benoit Baudry, KTH
Author(s)	:	Benjamin Danglot, INRIA Daniele Gagliardi, Engineering Yosu Gorronogoitia, Atos Caroline Landry, INRIA Oscar Luis Vera-Pérez, INRIA
Reviewer(s)	:	Vincent Massol, XWiki

Abstract

This deliverable reports on the final innovations and research results towards the construction of a tool-box for unit test amplification. This final, consolidated version of the tool-box can integrate in an automated build pipeline in order to provide relevant feedback to developers to improve the quality of industrial open source projects. The tool-box includes three main technical bricks: DSpot for unit test amplification; Descartes for test quality assessment; flaky-tool to analyze tests in the CI.

The last phase of the project focused on consolidating the DSpot and Descartes bricks. For DSpot we implemented novel algorithms to facilitate the integration in the build pipeline and support amplification on a commit. We also worked on performance analysis and improvement for DSpot in order to support the adoption in a industry-scale context. For Descartes, we developed novel analysis techniques to provide actionable feedback to developers about how they can address the issue of pseudo-tested methods.

KEYWORDS

Keyword List

Unit tests, test quality, test amplification, program analysis, program transformation

Revision History

Version	Type of Change	Author(s)
1.0	initial setup	Caroline Landry, INRIA
1.1	Descartes chapter	Oscar Luis Vera Perez, INRIA
1.2	Abstract, intro	Benoit Baudry, KTH
1.3	DSpot chapter	Benjamin Danglot, INRIA
1.4	Industrialization chapter	Daniele Gagliardi, ENG
1.5	DSpot optimization	Yosu Gorronogoitia, ATOS
1.6	Typos and review comments	Caroline Landry, INRIA
1.7	Updating Industrialization chapter after review	Daniele Gagliardi, ENG

Contents

1	Introduction	7
1.1	Relation to WP1 tasks	8
2	DSpot: tool for unit test amplification	9
2.1	Motivation & Background	9
2.1.1	Motivating Example	9
2.1.2	Behavioral Change	10
2.1.3	Behavioral Change Detection	10
2.1.4	Test Amplification	10
2.2	Behavioral Change Detection Approach	10
2.2.1	Overview of DCI	11
2.2.2	Test Selection and Diff Coverage	11
2.3	Evaluation	11
2.3.1	Benchmark	12
2.3.2	Protocol	13
2.3.3	Results	13
3	DSpot optimization	18
3.1	Motivation	18
3.2	Software profiling: tools and methods	19
3.3	Profiling findings	22
3.4	Performance optimization of DSpot	25
3.4.1	Techniques implemented	25
3.4.2	Results	26
3.5	Memory optimization of DSpot	27
3.5.1	Techniques implemented	27
3.5.2	Results	28
3.6	DSpot Optimization: overall experimentation	28
4	Descartes	32
4.1	Development progress	32
4.2	Generating automatic test improvement suggestions	32
4.2.1	Overview of the process for test improvement suggestions	33
4.2.2	Infection detection	33
4.2.3	Propagation detection	36
4.2.4	Suggestion generation	38
4.2.5	Implementation	40
4.2.6	Evaluation	40
4.3	Appendix	43



5	Industrialization	44
5.1	CI/CD solution selection	45
5.2	Building blocks	47
5.3	Industrial adoption	47
5.3.1	DSpot and Descartes execution in Blue Ocean	47
5.3.2	Descartes/PitMP integration test	48
5.3.3	Embedding DSpot and Descartes in Jenkins	50
5.3.4	DSpot execution optimization in CI	52
6	Conclusion	54
6.1	Publications for WP1	54
	Bibliography	55

Chapter 1

Introduction

This deliverable reports on the latest features and improvements towards the construction of a tool-box for unit testing amplification. This tool-box aims both at exploring novel research questions and at providing relevant feedback to developers to improve the quality of industrial open source projects. The core research question we address within workpackage 1 is as follows:

Can the automatic analysis of test and application code amplify the value of existing test suites in order to provide actionable hints for developers to make these test suites stronger?

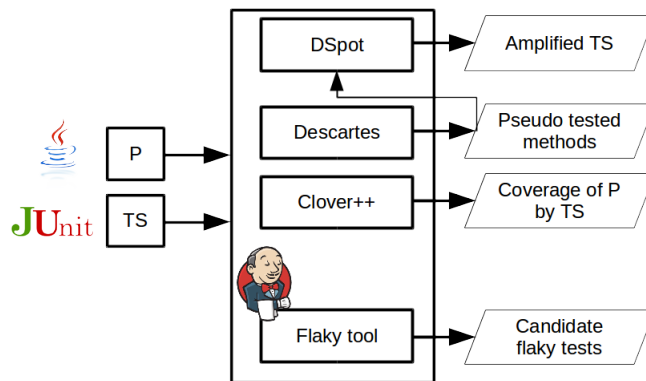


Figure 1.1: Overview of the test amplification tool-box features in the last period of STAMP

This deliverable reports on the tool-box that we have consolidated to address this original and hard question. As illustrated in Figure 1.1, we consider a program and a test suite as inputs for all our tools. A key innovation in the last period has been to integrate test improvements in the continuous integration engine and amplify tests with respect to a commit. The other key novelty has been to enhance Descartes with the capacity to suggest concrete actions to the developers in order to improve their test suite (beyond the list of pseudo-tested methods generated by the previous version of Descartes). The novel work developed in this period has led to two new outcomes of the test amplification toolbox

- proposals of improvements for the test cases that target the code pushed in a commit (as discussed in chapter 2)
- a detailed report that explains why a method is pseudo-tested and how the developer can take action to improve the test suite. chapter 4)

The last period has also been dedicated to the improvement of the solidity of the toolbox. We focused on

- Optimization of the performance of DSpot (chapter 2)
- Integration of DSpot and Descartes in the main build and CI tool chains (chapter 5).

The tools and results presented here are follow ups on the tools and results presented 9 months ago in deliverable D1.3. The key novelty lies are discussed above.

1.1 Relation to WP1 tasks

In Table 1.1 we summarize how the prototypes and results reported in this deliverable relate to the 5 tasks of WP1.

Table 1.1: WP1 tasks reported in this deliverable

Task 1.1	This task ended at M6 and is not covered by this deliverable
Task 1.2	We have consolidated the Descartes tool with advanced static and dynamic analyses that observe the interplay between a test suite and the program under test in order to generate actionable suggestions to the developers who seek to get rid of pseudo-tested methods (chapter 4)
Task 1.3	We have added a feature in DSpot to investigate the amplification of unit tests with respect to a specific commit (chapter 2)
Task 1.4	We have performed an in-depth analysis of the performance of DSpot and have improved the tool to reduce the resources needed for unit test amplification
Task 1.5	We have consolidated a set of tools to integrate amplification in standard DevOps pipelines (chapter 5)

Chapter 2

DSpot: tool for unit test amplification

DSpot is the key piece of software technology developed as part of WP1: a tool to investigate the novel concept of *unit test amplification*. This deliverable reports on the latest development of DSpot, focusing on the *integration of unit test amplification on a commit*. In the current period, we also improved the performance of DSpot as part of task T1.4.

2.1 Motivation & Background

Here, 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.

2.1.1 Motivating Example

On August 10, a developer pushed a commit to the master branch of the XWiki-commons project. The change¹, displayed in Listing 2.1, 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.

```

1 @@ -260,7 +260,8 @@ public boolean equals(Object object)
2 } else {
3 if (object instanceof FilterStreamType) {
4 result = Objects.equals(getType(), ((FilterStreamType) object).getType())
5 -    && Objects.equals(getDataFormat(),
6 -    ((FilterStreamType) object).getDataFormat());
7 +    && Objects.equals(getDataFormat(),
8 +    ((FilterStreamType) object).getDataFormat())
9 +    && Objects.equals(getVersion(),
10 +    ((FilterStreamType) object).getVersion());
11 } else {
12 result = false;
13 }

```

Listing 2.1: Commit 7E79F77 on XWiki-Commons that changes the behavior without a test.

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

¹<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. 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.

Below, we describe a complete scenario to sum up our vision of our approach's usage. A developer commits a change into the program. The Continuous Integration service is triggered; the CI analyzes the commit. There are two potential outcomes:

1) the developer provided a new test case or a modification to an existing one. In this case, the CI runs as usual, *e.g.*, it executes the test suite;

2) the developer did not provide a new test nor the modification of an existing one, the CI runs DSpot 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`[2].

2.1.2 Behavioral Change

A *behavioral change* is a source-code modification that triggers a new state for some inputs [5]. 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.

2.1.3 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. We propose a novel approach to detect behavioral changes based on test amplification.

2.1.4 Test Amplification

Test amplification is the idea of improving existing tests with respect to a specific test criterion [8]. 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. The test objective is to improve the test suite's detection of behavioral changes introduced by commits.

2.2 Behavioral Change Detection Approach

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

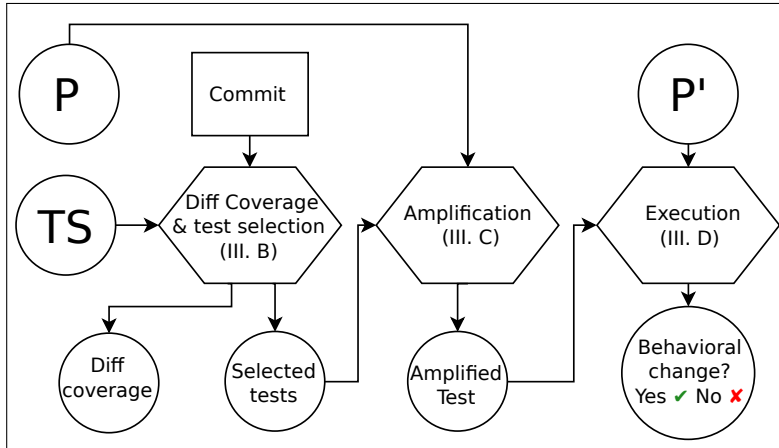


Figure 2.1: Overview of our approach to detect behavioral changes in commits.

2.2.1 Overview of DCI

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

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

DCI computes the code coverage of the diff and selects test methods accordingly. Then, it applies two kinds of test amplification to generate new test methods that detect the behavioral change. Figure 2.1 sums up the different phases of the approach:

- 1) Compute the diff coverage and select the tests to be amplified;
- 2) Amplify the selected tests based on the pre-commit version;
- 3) Execute amplified test methods against the post-commit version, and keep the failing test methods.

This process produces test methods that pass on the pre-commit version, fail on the post-commit version, hence they detect at least one behavioral change introduced by a given commit.

2.2.2 Test Selection and Diff Coverage

DCI implements a feature that: 1. reports the diff coverage of a commit, and 2. selects the set of unit tests that execute the diff. To do so, DCI first computes the code coverage for the whole test suite. Second, it identifies the test methods that hit the statements modified by the diff. Third, it produces the two outcomes elicited earlier: the diff coverage, computed as the ratio of statements in the diff covered by the test suite over the total number of statements in the diff and the list of test methods that cover the diff.

Then, 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.

2.3 Evaluation

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

- RQ1: To what extent are DSpot assertion and input amplifications able to produce amplified test methods that detect the behavioral changes?
- RQ2: What is the impact of the number of iteration performed by assertion amplification?

2.3.1 Benchmark

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

Project selection We need software projects that are

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

We pick the projects from the dataset in [6] and [1], which is composed of mature Java projects from GitHub.

Commit selection We take commits in inverse chronological order, from newest to oldest. We select the first ten commits that match the following criteria:

- 1) the commit modifies Java files (most behavioral changes are source code changes.²);
- 2) the commit provides or modifies a manually written test that detects a behavioral change. To verify this property, we execute the test on the pre-commit version. If it fails, it means that the test detects at least 1 behavioral change.
- 3) The changes of the commit must be covered by the pre-commit test suite. To do so, we compute the diff coverage. If the coverage is 0%, we discard the commit. We do this because if the change is not covered, we cannot select any test methods to be amplified, which is what we want to evaluate. Together, these criteria ensure that all selected commits:
 - 1) introduce behavioral changes,
 - 2) provide or modify a manually written test case that detects a behavioral change (which will be used as ground-truth for comparing generated tests), and
 - 3) that there is at least 1 test in the pre-commit version of the program that executes the diff and can be used to seed the amplification process.
- 4) There is no structural change in the commit between both versions, *e.g.*, no change in method signature and deletion of classes (this is ensured since the pre-commit test suite compiles and runs against the post-commit version of the program and vice-versa.)

Table 2.1: Considered Period for Selecting Commits.

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

²We are aware that behavioral changes can be introduced in other ways, such as modifying dependencies or configuration files [3].

Final benchmark Table 2.1 shows the main statistics on the benchmark dataset. The first column is the name of the considered project; The second column is the number of lines of code computed with cloc; The third column is the date of the oldest commit for the project; The fourth column is the date of the newest commit for the project; The fifth, sixth and seventh are respectively the total number of commit we analyze, the total number of commits we discard, the number of commits that match all our criteria but the third (there is no test in the pre-commit that execute the change and the number of commit we select). The last row reports a summary of the benchmark with the total number of lines of codes, the oldest and the newest dates, the average number of commits analyzed, the average number of commits discarded, the average number of commits matching all the criteria but the third. 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>.

2.3.2 Protocol

To answer **RQ1**, we run DSpot’s assertion and input amplifications separately 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 the assertion amplification 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.

2.3.3 Results

The overall results are reported in Table 2.2. The first column is the shortened commit id; the second column is the commit date; the third column is the total number of test methods executed when building that version of the project; the fourth and fifth columns are respectively the number of tests modified or added by the commit, and the size of the diff in terms of line additions (in green) and deletions (in red); the sixth and seventh columns are respectively the diff coverage and the number of tests DCI selected; the eighth column provides the amplification results for input amplification, and it is either a ✓ with the number of amplified tests that detect a behavioral change or a 0 if DCI did not succeed in generating a test that detects a change; the ninth column displays the time spent on the amplification phase; The tenth and the eleventh are respectively a ✓ with the number of amplified tests for DCI_{SBAMPL} (or 0 if a change is not detected) for 3 iterations.

RQ1: To what extent are DSpot assertion and input amplifications able to produce amplified test methods that detect the behavioral changes?

We now focus on the last 4 columns of Table 2.2. For instance, for COMMONS-IO#F00D97A (4th row), assertion amplification generated 39 amplified tests that detect the behavioral change. For COMMONS-IO#81210EB (8th row), only the Input-amplification version of DCI detects the change. Overall, using only Assertion-amplification, DCI generates amplified tests that detect 9 out of 60 behavioral changes. Meanwhile, using Input-amplification only, DCI generates amplified tests that detect 28 out of 60 behavioral changes.

Regarding the number of generated tests, input amplification generates a large number of test cases, compared to assertion amplification only (15 versus 6708, see column “total” at the bottom of the table). Both strategies can generate amplified tests, however since assertion amplification does not produce a large amount of test methods the developers do not have to triage a large set of test cases. Also, since assertion amplification only adds assertions, the amplified tests are easier to understand than the ones generated by input amplification

Table 2.2: Performance evaluation of DCI on 60 commits from 6 large open-source projects.

	id	date	#Test	#Modified Tests	+ / -	Cov	#Selected Tests	#Assertion-amplification Tests	Time	#Input-amplification 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
commons-lang	f56931c	7/2/18	4105	1	30 / 4	25.0	42	0	2.4m	✓ (1)	4.4h
	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
	50ce8e4	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	142	8.3h
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
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
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
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
	adfeec	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
total							3378	9(15)	2.2h	26(6708)	100.9h

Overall, DCI successfully generates amplified tests that detect a behavioral change in 46% of the commits in our benchmark (28 out of 60). Recall that the 60 commits that we analyze are real changes that fix bugs in complex code bases. They represent modifications, sometimes deep in the code, that represent challenges with respect to testability [7]. Consequently, the fact DCI can generate test cases that detect behavioral changes, is considered an achievement. The commits for which DCI fails to detect the change can be considered as a target for future research on this topic.

Now, we manually analyze a successful case where DCI detects the behavioral change. We select commit 3FADFDD³ from commons-lang, which is succinct enough to be discussed. The diff is shown in Listing 2.2.

```

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 2.2: Diff of commit 3FADFDD from commons-lang.

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

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

```

1 @Test
2 public void testPerson() {
3     final Person p = new Person();
4     p.name = "Jane Doe";
5     p.age = 25;
6     p.smoker = true;
7     assertEquals("{\"name\":\"Jane Doe\",\"age\":25,\"smoker\":true}",
8         new ToStringBuilder(p).append("name", p.name)
9             .append("age", p.age).append("smoker", p.smoker)
10             .toString()
11 );
12 }

```

Listing 2.3: Seed test method selected to be amplified for commit 3FADFDD from commons-lang.

The test, shown Listing 2.3 is selected for amplification because it triggers the execution of the changed line.

```

1 @Test(timeout = 10000)
2 public void testPerson_literalMutationString85602() throws Exception {
3     final ToStringStyleTest.Person p = new ToStringStyleTest.Person();
4     p.name = "Jane Doe";
5     Assert.assertEquals("Jane Doe", p.name);
6     p.age = 25;
7     p.smoker = true;
8     String o_testPerson_literalMutationString85602__6 = new ToStringBuilder(p)
9         .append("n/me", p.name)
10        .append("age", p.age)
11        .append("smoker", p.smoker)
12        .toString();
13     Assert.assertEquals(
14         "{\"n/me\":\"Jane Doe\",\"age\":25,\"smoker\":true}",
15         o_testPerson_literalMutationString85602__6

```

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

```

16 );
17 Assert.assertEquals("Jane Doe", p.name);
18 }

```

Listing 2.4: Test generated by DCI that detects the behavioral change of 3FADFDD from commons-lang.

We show in Listing 2.4 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 detects 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).
```

RQ2: What is the impact of the number of iteration performed by assertion amplification?

The results are reported in Table 2.3. This table can be read as follow: the first column is the name of the project; the second column is the commit identifier; then, the third, fourth, fifth, sixth, seventh and eighth provide the amplification results and execution time for each number of iteration 1, 2, and 3. A ✓ indicates with the number of amplified tests that detect a behavioral change and a - denotes that DCI did not succeed in generating a test that detects a change.

Overall, input amplification generates amplified tests that detect 21, 23, and 24 out of 60 behavioral changes for respectively $iteration = 1$, $iteration = 2$ and $iteration = 3$. The more iteration, the more DSpot explores, the more it generates amplified tests that detect the behavioral changes but the more it takes time also. When amplifying inputs with $iteration = 3$, DSpot generates amplified test methods that detect 3 more behavioral changes than when it is used with $iteration = 1$ and 1 then when it is used with $iteration = 2$. It represents an increase of 14% and 4% for respectively $iteration = 1$ and $iteration = 2$.

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

Table 2.3: Evaluation of the impact of the number of iteration done by DCI_{SBAMPL} on 60 commits from 6 open-source projects.

	id	$it = 1$	Time	$it = 2$	Time	$it = 3$	Time
commons-io	c6b8a38	0	25.0s	0	62.0s	0	98.0s
	2736b6f	✓(1)	26.1m	✓(2)	44.2m	✓(12)	76.3m
	a4705cc	0	4.1m	0	21.1m	0	38.1m
	f00d97a	✓(7)	13.0s	✓(28)	19.0s	✓(39)	27.0s
	3378280	✓(6)	15.0s	✓(10)	20.0s	✓(11)	24.0s
	703228a	0	30.3m	0	55.1m	0	71.0m
	a7bd568	0	28.6m	0	52.0m	0	65.2m
	81210eb	✓(2)	14.0s	✓(4)	18.0s	✓(8)	23.0s
	57f493a	0	20.0s	0	32.0s	0	54.0s
	5d072ef	✓(461)	32.2m	✓(1014)	65.5m	✓(1538)	2.2h
	total	47.70	avg(12.3m)	105.80	avg(24.0m)	160.80	avg(38.8m)
commons-lang	f56931c	0	0.0s	0	3.7m	✓(0)	8.3m
	87937b2	0	3.5m	0	10.5m	0	18.1m
	09ef69c	0	97.0s	0	21.0m	0	98.8m
	3fadfdd	✓(1)	2.0m	✓(1)	9.3m	✓(4)	17.2m
	e7d16c2	✓(3)	111.0s	✓(2)	8.4m	✓(2)	15.1m
	50ce8c4	✓(61)	38.0s	✓(97)	78.0s	✓(135)	2.0m
	2e9f3a8	0	11.4m	0	35.0m	0	66.5m
	c8e61af	0	16.0s	0	16.0s	0	16.0s
	d8ec011	0	36.0s	0	68.0s	0	2.3m
	7d061e3	0	79.0s	0	5.8m	0	11.4m
	total	6.50	avg(2.3m)	10.00	avg(9.6m)	14.10	avg(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	1.10	avg(13.6m)	7.30	avg(46.0m)	15.30	avg(86.5m)
jsoup	426ffe7	✓(126)	5.4m	✓(172)	19.2m	✓(198)	33.6m
	a810d2e	0	90.0s	0	13.9m	0	26.6m
	6be19a6	0	8.1m	0	39.7m	0	67.7m
	e38dfd4	0	117.0s	0	6.3m	0	12.5m
	e9feec9	0	20.0s	0	50.0s	0	95.0s
	0f7e0cc	✓(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	13.30	avg(19.4m)	21.40	avg(69.8m)	28.60	avg(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	32.60	avg(84.3m)	160.10	avg(2.5h)	442.20	avg(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
	adfeec	✓(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	4.50	avg(10.3m)	9.00	avg(29.7m)	9.80	avg(49.5m)
	total	22(18.12)	avg(23.7m)	23(52.56)	avg(54.9m)	25(114.76)	avg(100.9m)

Chapter 3

DSpot optimization

3.1 Motivation

The optimization of the JUnit test amplification process is one of the objectives of T1.4. In particular, this task aims at developing a “a test runner that speeds up the execution of amplified test suites, exploiting knowledge about the commonalities between test case”. The ultimate reason for this test runner is to speed up the overall JUnit test amplification process: a computational intensive task, which requires the repetitive execution of a large number of generated amplified tests, before they are judged to be discarded or accepted as valid, according to some criteria such as the code coverage amplification.

JUnit test amplification process is performed by DSpot [1] tool, which has been developed from scratch in the context of the STAMP project. DSpot users have reported several issues related to DSpot performance and memory consumption. This is the main reason that DSpot is considered the main target of the optimization activity conducted in T1.4.

Descartes could benefit from a similar optimization refactoring. However, Descartes is implemented as a PITest plugin, which already benefits from the PITest optimizations, such as the parallel test and JUnit common test suite preamble executions. Besides, Descartes overall performance has been considered acceptable in most of use case experiments since industrial evaluators have not posted issues about Descartes performance. For this reason, Descartes has not been included in the T1.4 optimization refactorings.

In the following, we will summarize the main experimental issues related to the DSpot performance and memory consumption factors, reported by industrial evaluators and scientific developers during the evaluation period that overlapped with the DSpot development lifecycle.

Performance related issues:

- Recursive loops: #444
- Thread deadlocks: #711
- Timeout exceptions: #425, #628
- Long execution times: #386
- DSpot mutation algorithm related optimizations: #691

Memory related issues:

- Stackoverflow issues: #444
- Out of memory: #617

Industrial evaluation reported in D5.5 [D5.5] and D5.6 [D5.6] also revealed some of the DSpot performance and memory consumption issues, as reported by evaluators who could run DSpot in their UCs.

Atos UC:

- “For some IF target test cases, DSpot crashes raising memory exceptions (OutOfMemory, StackOverflow) that cannot be sorted out by increasing the JVM memory”,
- “In other test cases, timeouts were reported. They were fixed by increasing the timeout parameter in DSpot input configuration”.

But it is also reported as conclusion:

- “performance: in previous evaluation, DSpot executions took over more than 12 hours to complete with a single run. The new releases evaluated in this period run over the same IF target test cases for no more than 3 hours in the more time consuming case”.

XWIKI UC:

- “When we tried using several amplifiers, the tests took very long (over 4 hours in the tests we did) and we had to stop them”,
- “The real interest of DSpot is that it doesn't require human intervention to generate new tests (as opposed to Descartes for example). Thus, if we're able to make DSpot run faster in the future, the idea will be to integrate it into our CI to automatically whenever code and tests are committed and to commit the generated tests”

OW2 UC:

- “Execution time and volumetry : During this period, our experimentations were only based on the compiled sources, using the command line to execute each time. But we do have in mind that those tools have the purpose to be integrated in the CI in the end. So execution time is of paramount importance in the CI process, it should not prevent a new release”.

As mentioned by some UCs, DSpot execution is primarily intended to be conducted within automated CI/CD pipelines or jobs, after new commits are pulled from the SCM repositories of the DSpot target software project. Thus, the overall CI/CD building, testing and delivery process should not take high computational resources (e.g. CPU computation time). That is, DSpot execution time and resources consumption should be restrained.

3.2 Software profiling: tools and methods

In order to identify the root cause of high computing resources consumption (i.e. CPU, memory) in DSpot, we first analyse its performance and memory usage. This requires a deep profiling analysis of the DSpot execution against a target test suite project.

Profiling is the process that collects execution statistics (CPU usage, threads, allocated/consumed memory, etc.) from a running process during a specific time frame. Statistics sampling rate is customized to balance the amount and precision of collected data without overloading the process execution time and the consumed computational resources. These statistics may largely help us to identify DSpot performance bottlenecks and memory leaks in DSpot modules. Found issues will be considered as the main targets for our performance improvement strategies.

Several industrial profilers are available for JVM processes. In DSpot performance analysis we evaluated and used the following:

JProfiler¹ is one of the best and more popular commercial profiling tools for JVM, but temporary available as it only offers a 10-days evaluation. During this evaluation period we could conduct the first profiling experiments on DSpot. JProfiler supports to profile a process starting it from a script. This starting feature is quite useful to have a complete profile of the entire DSpot execution. JProfiler offers an overview particularly interesting to visualize the CPU and Memory overall telemetries. The heap view gives us statistics about object (by type) allocation, ordered by size. We use this view to identify the object types most contributing to the memory pressure. The CPU view gives us the CPU time consumed (in percentage over the total) in every frame (e.g. method invocation) with the thread execution stack. JProfiler instruments the DSpot executed code so that it can take (quite efficiently) precise statistics about the CPU time consumed by each method within the main thread (but also in others) frame stack. Given a particular method frame, it computes the high hot spots, that is, the most CPU consuming nested methods invoked within that method call. This information is quite useful to identify performance bottlenecks.

Oracle Java Mission Control (JMC)² is shipped within the JDK since Java 7. It is a sampling-based (not instrumented) extremely fast profiling tool. It also offers memory and cpu views as JProfiler, but it faces a remarkable limitation compared to it, as it only samples CPU usages, compiling a sampling map that shows the methods mostly visited by the CPU during the DSpot execution: typically the deepest nested methods in the execution call stack. However, it neither instrument the running code nor aggregate the CPU samples within the top calling methods, so it neither identify main bottleneck hot spots nor offer real statistics about CPU usage by each method, including the inner invocations to nested methods. As a consequence, the results obtained with JVC didn't helps us to identify DSpot performance bottlenecks. Another limitation is caused by the fact the JMC cannot start DSpot from a launching script, but it has to be captured from the JMC console by hand, once DSpot is already running. Moreover, JMC profiling needs to be manually stopped before the target DSpot VM quits, otherwise the profiling data is lost. These issues impede to profile the entire DSpot execution and makes difficult to select a profiling time slot. The memory view offered similar information compared to the one provided by JProfiler.

Particularly interested was the thread status information provided by the event view that enabled us to understand the reasons the main thread was idle waiting for other threads to return the execution control (for instance during the amplified test execution).

Netbeans profiler³ is a full-fledged profiler shipped within the Netbeans IDE. It is similar in features to JProfiler but without the commercial usage restrictions so that it was the main profiling tool we used for detecting DSpot performance bottlenecks. Netbeans profiler instruments DSpot code and provides precise aggregated method call statistics that report hop spots that helps us to identify bottlenecks. Concerning memory leak detection, Netbeans profiler telemetry view shows statistics about surviving generations. A high number of surviving generations may suggest a memory leak. A memory generation is the group of objects allocated in the period between two garbage collector (GC) invocations. A surviving generation is group of allocated objects still leaving after a GC invocation, so if this number grows unbound, it may imply some objects not eligible to be disposed off by the GC.

Eclipse Memory Analyzer (MAT)⁴ is an Eclipse-based, very powerful analyzer of Java memory dumps. We obtained these dumps from DSpot at a particular execution point using tools such as JMap⁵ or JCmd⁶. Among its different analysis profiles and views, MAT gives an overview of detected memory problem suspects: candidates for memory leaks. Using its shortest path to accumulation point we can spot the DSpot code object that holds the memory reference causing the leak. MAT also

¹<https://www.ej-technologies.com/products/jprofiler/overview.html>

²<https://www.oracle.com/technetwork/java/javaseproducts/mission-control/java-mission-control-1998576.html>

³<https://profiler.netbeans.org/>

⁴<https://www.eclipse.org/mat/>

⁵<https://docs.oracle.com/javase/7/docs/technotes/tools/share/jmap.html>

⁶<https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/tooldescr006.html>

offers a powerful SQL-like query language to find and count memory objects by type.

Before we opted for MAT, we also analyze DSpot memory dumps using online memory analyzers such as HeapHero⁷, but we discarded it as the main memory analysis tool because its limitations to analyse the big DSpot dumps. Nonetheless, HeapHero reports was quite useful to identify some of the DSpot detected memory leaks.

We profiled the DSpot execution using these tools along the entire optimization process. Initially using JProfile and JMC, obtaining useful insights, and later on using Netbeans profiler and MAT during a larger profiling period. We adopted an iterative profiling process using DHell project as target with different configurations (mostly consisting of permuting the selector among Jacoco and PIT and changing the number of iterations, the list of amplifiers, over the entire DHell test suite). The iterative process went through the following tasks:

1. Profiling the DSpot execution for a given input configuration, using one of the above profilers,
2. Perform a detailed analysis of profiling results,
3. Identify target performance bottlenecks and/or memory issues from this analysis,
4. Analyse the causes of detected target bottleneck or memory issues,
5. Design code fixes for DSpot and/or bound libraries (i.e. TestRunner) code,
6. Implement the fix patches,
7. Create push requests in DSpot GitHub repository pointing at these code fixes,
8. Profile DSpot execution, using the same input configuration tested before, in order to validate the usefulness of the implemented patch to fix the detected bottleneck/memory issue,
9. Reports results in the PR,
10. Request the acceptance of the PR.

This iterative process was conducted several times. As a result, 7 key performance improvements were implemented and merged in DSpot. Each of these improvements are discussed below. The results of this iterative analysis, the detected performance bottlenecks and memory issues, the implemented patches and its results on performance and memory consumption are reported in the following sections.

In order to run DSpot with profilers and memory analyzers, a number of launching scripts were created:

- JProfiler requires to run DSpot JVM with this flag: `-agentpath:/path-to-jprofiler10/bin/linux-x64/libjprofilerti.so=port=8849`
- Java Mission Control requires to run DSpot JVM with this flag: `-XX:+UnlockCommercialFeatures -XX:+FlightRecorder -XX:+UnlockDiagnosticVMOptions -XX:+DebugNonSafepoints`
- Netbeans Profiler requires to run DSpot JVM with this flag: `-agentpath:/path-to-netbeans/profiler/lib/depoyed/jdk16/linux-amd64/libprofilerinterface.so=/home/yosu/Development/netbeans/profiler/lib,5140`
- To capture DSpot memory dumps using Jcmd, the following script was used: `jcmd $1 GC.heap_dump /path-to-dump/dump.bin <where $1 is DSpot PID>`
- To capture DSpot memory dumps using Jmap, the following script was used: `jmap -dump:live,format=b,file=/path-to-dump/dump.bin $1 <where $1 is DSpot PID>`

⁷<https://heaphero.io>

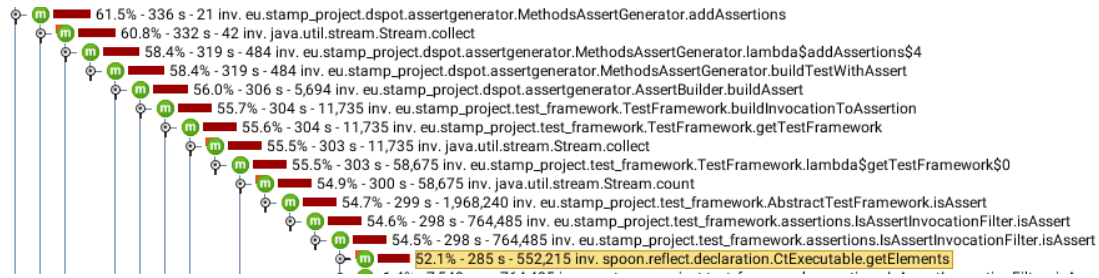


Figure 3.1: DSpot execution JProfiler statistics showing TestFramework CPU consumption

eu.stamp_project.utils.DSpotUtils.printAndCompileToCheck (spoon.reflect.declaration.CtType, java.io.File)	860,598 ms (55.7%)	860,598 ms (74.1%)
java.util.stream.ReferencePipeline.forEach (java.util.function.Consumer)	385,838 ms (25%)	385,838 ms (33.2%)
eu.stamp_project.utils.compilation.DSpotCompiler.compile (eu.stamp_project.utils.program.InputConfiguration, String, String)	258,840 ms (16.8%)	258,840 ms (22.3%)
eu.stamp_project.utils.DSpotUtils.printCtTypeToGivenDirectory (spoon.reflect.declaration.CtType, java.io.File, boolean)	113,177 ms (7.3%)	113,177 ms (9.7%)
eu.stamp_project.utils.DSpotUtils.getExistingClass (spoon.reflect.declaration.CtType, String)	102,674 ms (6.7%)	102,674 ms (8.8%)

Figure 3.2: DSpot execution Netbeans profiler statistics showing printAndCompileToCheck CPU consumption

3.3 Profiling findings

As a result of the profiling analysis process described above, the following findings, classified onto performance bottlenecks, and memory issues (leaks, suspects, etc), were found.

Performance bottlenecks

Issue 1: eu.stamp_project.test_framework.TestFramework.getTestFramework

DSpot spends in this method 55.6% of the total execution time (see Figure 3.1), and it is invoked over 11.700 times. It computes the test framework (e.g. JUnit4, JUnit5) associated to an amplified test class. However, this method is pure functional (i.e. invariant) for all amplified clones of the same input test class. Therefore, it is redundant to invoke this method repetitively.

Issue 2: eu.stamp_project.utils.DSpotUtils.printAndCompileToCheck

Within this method, DSpot spends 25% of total execution time on transversing the stream of methods contained within an amplified test class (see Figure 3.2), using a for-each loop that relies on a filter predicate that invokes the method CtClass.getMethods of the Spoon library⁸. This method is invariant, returning always the same collection.

Issue 3: eu.stamp_project.dspot.budget.NoBudgetizer.inputAmplify

Another performance bottleneck was detected in the NoBudgetizer.inputAmplify method, which takes over 13% of the total execution time when Jacoco selector is used and over 20% when PIT selector is (see Figure 3.3). This method invokes the NoBudgetizer.reduce method, which keeps a configurable maximum number of amplified methods, and discards the remaining ones. This is done by a filter that sorts the amplified tests according to the textual distance between among them, raking higher the most distant to the others. This filter needs to compute the textual representation of the method by invoking its CtMethod.getString and this takes most of the computational time.

Issue 4: eu.stamp_project.utils.compilation.TestCompiler.compileAndRun

Most of the DSpot execution time (over 49%) is spent on running the amplified tests (see Figure 3.4) as well on selecting among them, those that better contribute to increase the coverage, by analysing them with the specified selector (e.g. Jacoco, PIT, etc). DSpot uses the associated test framework (e.g. JUnit4 or 5) to run these tests, but execution runs sequentially: the CPU total execution time grows with the number of executed tests, hence.

⁸Spoon library (<http://spoon.gforge.inria.fr/>) is intensively used by DSpot to manage a memory model of the target Java test and source classes.

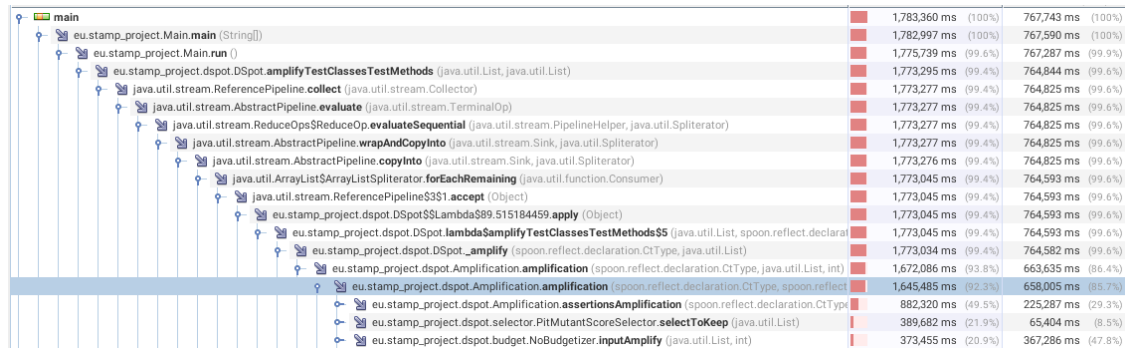


Figure 3.3: Netbeans profiler CPU consumption for amplification method, using PIT selector, showing NoBudgetizer.inputAmplify statistics.

Name	Total Time	Total Time (CPU)
eu.stamp_project.utils.compilation.TestCompiler.compileAndRun (spoon.reflect.declaration.CtType, eu.stamp_project.utils.compilation.DSpotCompiler, java.util.List, java.util.List)	639,329 ms (54.5%)	51,501 ms (11.6%)
eu.stamp_project.utils.execution.TestRunner.runGivenTestMethods (spoon.reflect.declaration.CtType, java.util.List, String)	576,576 ms (49.1%)	600 ms (0.1%)
eu.stamp_project.utils.execution.TestRunner.run (String, String, String, String)	576,551 ms (49.1%)	575 ms (0.1%)

Figure 3.4: Netbeans profiler CPU consumption statistics for compileAndRun method.

Issue 5 DSpot thread deadlock

Investigating DSpot performance using profiling it was detected a possible thread deadlock reported in #711. The reason behind this thread deadlock is the way DSpot forks another thread to actually run the amplify tests: `EntryPoint.runGivenCommandLine` forks (in another JVM) the execution of the test cases using an object of the `eu.stamp_project.testrunner.runner.JUnit4Runner` class, which is contained within the `TestRunner` library Dspot relies on. This class uses a discourage, old-style `Runtime.getRuntime().exec` method to fork the runner. Next, an additional thread is forked within the DSpot JVM to capture `TestRunner` JVM logs to be shown to the user. Finally, the DSpot main thread waits for `TestRunner` JVM to notify it has completed the test execution before proceeding. But `TestRunner` thread never ends, causing the main thread in DSpot to keep blocked for ever.

Issue 6 Test parallel execution: test results serialization issues

When amplified tests are executed in parallel, serialization issues are detected. They are caused by the simultaneous way different threads executing the tests are attempting to access the test result objects.

Memory issues

Issue 1: Increasing number of surviving generations

Memory profiling shows an increasing number of surviving generations in memory allocation, which are not released by the GC after being executed (cf. Figure 3.5). This number grows further when we increase the number of DSpot iterations. This is a sign of a memory leak that needs to be further investigated.

Issue 2: eu.stamp_project.utils.AmplificationHelper collections

This helper holds different collections to keep track of the cloned tests methods (i.e. `Spoon CtMethodImpl` objects) back to the original ones. Initially, `HeapHero` report spotted a potential memory issue related to the `ampTestToParent` (Figure 3.6), which took over 93% of total DSpot allocated memory, containing over 2.500 references to `tMethodImpl` objects. However, most of these methods objects are disposed of by DSpot once they are discarded, but since their references are not removed from this collection, it causes a memory leak. This leak explain the increasing number of surviving generations mentioned in previous issue.

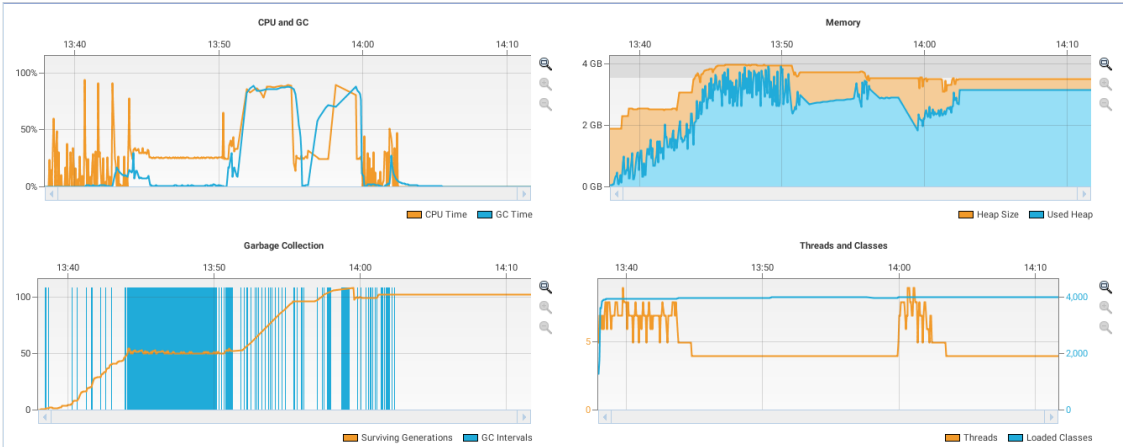


Figure 3.5: Netbeans telemetry view showing an increasing number of surviving generations (left-bottom graph)


Name	Percentage	Size
 <code>Java Static eu.stamp_project.utils.AmplificationHelper.ampTestToParent</code> ↗ «Object might be causing memory leak»	93.4%	981.39mb
<code>Java Static eu.stamp_project.utils.program.InputConfiguration.instance</code> ↗	3.5%	36.38mb
<code>Java Static eu.stamp_project.utils.AmplificationHelper.originalTestBindings</code> ↗	1.4%	15.19mb
<code>Java Local@6c0081220 (eu.stamp_project.dspot.Amplification)</code> ↗	0.8%	8.3mb
<code>Java Static eu.stamp_project.utils.Counter.instance</code> ↗	0.2%	2.24mb
... and 6251 more objects retaining 4.86mb (0.5%)		

Figure 3.6: HeapHero memory report showing ampTestToParent memory allocated

Class Name	Shallow Heap	Retained Heap	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
sun.misc.Launcher\$AppClassLoader @ 0x6c0008ed0	88	3,372,625,288	84.11%
java.util.Vector @ 0x6c0184d00	32	3,372,446,088	84.11%
java.lang.Object[2560] @ 0x6c0184d20	10,256	3,372,446,056	84.11%
class eu.stamp_project.utils.AmplificationHelper @ 0x6c00a7790	40	3,343,701,264	83.39%
java.util.IdentityHashMap @ 0x6c0350aa8	40	3,153,346,624	78.65%
class eu.stamp_project.utils.AmplificationHelper @ 0x6c00a7790	40	3,343,701,264	
<class> class java.lang.Class @ 0x6c0005ab8 System Class	40	1,264	
<super> class java.lang.Object @ 0x6c0005918 System Class	8	256	
<classloader> sun.misc.Launcher\$AppClassLoader @ 0x6c0008ed0	88	3,372,625,288	
<resolved_references> java.lang.Object[66] @ 0x6c0029d50	280	872	
LINE_SEPARATOR java.lang.String @ 0x6c008a540 \u000a	24	48	
PATH_SEPARATOR java.lang.String @ 0x6c0256b88 :	24	48	
ampTestToParent java.util.IdentityHashMap @ 0x6c0348de0	40	89,838,792	
importByClass java.util.HashMap @ 0x6c0348e08	48	48	
LOGGER org.slf4j.impl.Log4jLoggerAdapter @ 0x6c034f4c0	24	24	
originalTestBindings java.util.IdentityHashMap @ 0x6c0350aa8	40	3,153,346,624	

Figure 3.7: MAT memory report showing originalTestBinding memory allocated

After the performance bottleneck issue 1 was fixed, MAT report also spotted a similar memory leak related to the collection `originalTestBindings`, introduced by the issue1 fix (Figure 3.7). This collection took over 78% of the total memory allocated by DSpot. It tracked references of test methods cloned during DSpot execution back to the original ones in order to retrieve their associated test framework. Its memory leak behavior was similar to `ampTestToParent` behavior.

3.4 Performance optimization of DSpot

In order to fix the performance issues detected by profiling (reported in previous section) we implemented a number of DSpot code patches (and new features) that were proposed as pull requests (PRs). The following described them technically.

3.4.1 Techniques implemented

Optimization 1: `eu.stamp_project.test_framework.TestFramework.getTestFramework`

This optimization relies on the fact that the test framework associated to a test method is the same than the one associated to its original test method, despite the amplifications performed. First, this optimization maintains a map that, for each cloned test method, associates it to its original test method. Next, when the test framework for a method is requested, it is retrieved from a cache that uses its original test method (recovered from the map) as a key. If the test framework is not found in the cache (i.e. this is the case of the first request for an original test case) it is computed and stored in the cache. This patch uses the EhCache⁹ library for an in-memory efficient cache implementation, preconfigured to store up to 104 method objects. This optimization was released in PR #701 and PR #735

Optimization 2: `eu.stamp_project.utils.DSpotUtils.printAndCompileToCheck`

This optimization extract out the invocation to the `CtType.getMethods` method from the stream filter applied to the methods of the amplified test class in `DSpotUtils.printAndCompileToCheck` method.

This optimization was released in PR #719

⁹<https://www.ehcache.org/>

Optimization 3: RandomBudgetizer

This optimization creates a `RandomBudgetizer` that randomly selects amplified methods to be kept until is reached a maximum number. This budgetizer is configured as the DSpot default, displacing the existing `NoBudgetizer` (it selects methods based on their mutual textual distance), which is renamed as `TextualDistanceBudgetizer`. DSpot input configuration has been extended to use this new budgetizer:

```
[--budgetizer <RandomBudgetizer | TextualDistanceBudgetizer |
SimpleBudgetizer>]
[optional] specify a Budgetizer.
Possible values are:
- RandomBudgetizer
- TextualDistanceBudgetizer
- SimpleBudgetizer
(default: RandomBudgetizer)
```

This optimization was released in PR [#774](#)

Optimization 4: Parallel test execution

This optimization implements JUnit4/5 support for test execution in DSpot. It runs in parallel most of tests executed by DSpot, excepting the instrumented tests and the PIT execution. Both test execution approaches supported by DSpot, namely, CMD-based and Maven-based are extended to support parallel execution.

JUnit4 parallel execution uses the `com.googlecode.junittoolbox.ParallelRunner` test runner, a JUnit runner included in the `junit-toolbox` library. The parameterization of the parallel execution (e.g. number of threads to use) is injected in the maven-surefire-plugin configuration in the generated `pom.xml` file.

JUnit5 parallel execution uses the JUnit5 `@Execution(ExecutionMode.CONCURRENT)` annotation. A number of JUnit5 additional libraries needs to be added to the classpath or the Maven `pom.xml` overall dependencies as well as in the maven-surefire-plugin. The parallel execution configuration is placed in a `junit-platform.properties` file located in the classpath.

DSpot input configuration has been extended to enable parallel test execution (disabled by default):

```
[--execute-test-parallel-with-number-processors
<execute-test-parallel-with-number-processors>]
[optional] If enabled, DSpot will execute the tests in parallel.
For JUnit5 tests it will use the number of given processors
(specify 0 to take the number of available core processors).
For JUnit4 tests, it will use the number of available CPU
processors (given number of processors is ignored). (default: 0)
```

This optimization was released in PR [#775](#) and PR [#815](#)

3.4.2 Results

DSpot experiments were conducted using Dhell project as test amplification target.

Optimization 1: `eu.stamp_project.test_framework.TestFramework.getTestFramework`

In experiments (3 iterations with default PIT selector, using `MethodAdd:AllLiteralAmplifiers:MethodGeneratorAmplifier` amplifiers, over the entire Dhell test suite), this optimization showed a significant overall test execution time reduction on the range [42%, 45%]. This is a consequence of releasing the high computational cost of computing

the test framework (using the Spoon in-memory model of the target test suite) for each of the thousand cloned test methods generated by DSpot in a common experiment.

Optimization 2: `eu.stamp_project.utils.DSpotUtils.printAndCompileToCheck`

This quite simple optimization showed a huge impact on DSpot overall performance, which was improved up to a 30% reduction on the total execution time. Profiling analysis after this optimization showed that this method still spent up to 75% of the total execution time, therefore it was the subject of further optimizations.

Optimization 3: `RandomBudgetizer`

In experiments (3 iterations using `MethodAdd:AllLiteralAmplifiers:MethodGeneratorAmplifier` amplifiers, over the entire Dhall test suite) overall execution time improvement ranges from 6% to 33%, for Jacoco and PIT selectors, respectively.

Optimization 4: `Parallel test execution`

Experiments (3 iterations, using `MethodAdd:AllLiteralAmplifiers:MethodGeneratorAmplifier` amplifiers, over the entire Dhall JUnit4 test suite, with `-execute-test-parallel-with-number-processors 4`) showed modest improvements on the overall execution time, in the range [2%-9,9%] and 14,4% for Jacoco and PIT selectors, respectively. This unexpected result is justified by the low Dhall test execution times (around few milliseconds each) so that they cannot take advantage of parallel execution (the overhead required to prepare the tests execution is higher than the actual time required to execute them). When we artificially introduced some delays (100-200 ms) into each test case, the performance improvement with parallel execution is significant: 31,87% for Jacoco selector, and 19,55% for PIT selector. Overall DSpot execution also depends (for JUnit5) on the number of threads configured for test parallel execution, so the optimal number needs to be determined by the experimenter. In case of JUnit4, the number of used threads for parallel test execution seems not to influence the overall DSpot execution time. Indeed the optimal number seems to be the total number of available CPU processors (set as default), so the configuration of thread has been disabled.

3.5 Memory optimization of DSpot

Similarly, memory issues detected by profiling (reported in previous section) were fixed by a number of DSpot code patches (and new features) proposed as pull requests (PRs). The following described them technically.

3.5.1 Techniques implemented

This optimization manages the `AmplificationHelper` dictionaries that hold references of cloned `CtMethods` through their parents up to the original ones. It also introduces some memory optimization to enable the GC to dispose of the cloned `CtMethods` that are not anymore required in the scope, particularly before the amplified class is serialized to the output folder.

This optimization uses the `WeakIdentityHashMap` from the Apache CXF¹⁰ common utilities library to keep weak references to cloned `CtMethods`, so they can be disposed of by the GC when they are discarded by DSpot. This is the way `AmplificationHelper` declares collections `ampTestToParent` and `originalTestBindings`.

In the main DSpot amplification loop, the intermediate placeholder that contains the amplified test methods is nullified, so they can be disposed of by the GC without waiting for the main loop to complete, so the number of surviving generations drops down significantly. This optimization was released in PR #736

¹⁰<https://cxf.apache.org/>



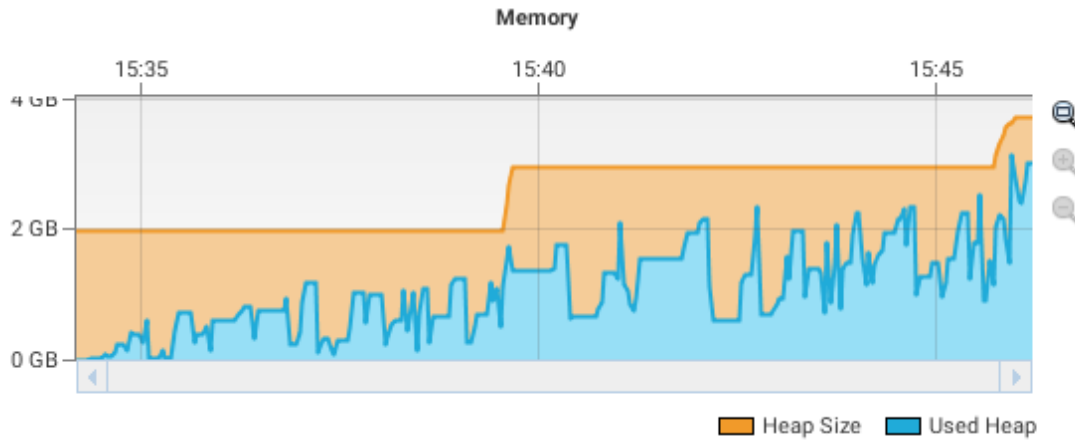


Figure 3.8: Memory consumption before memory optimization

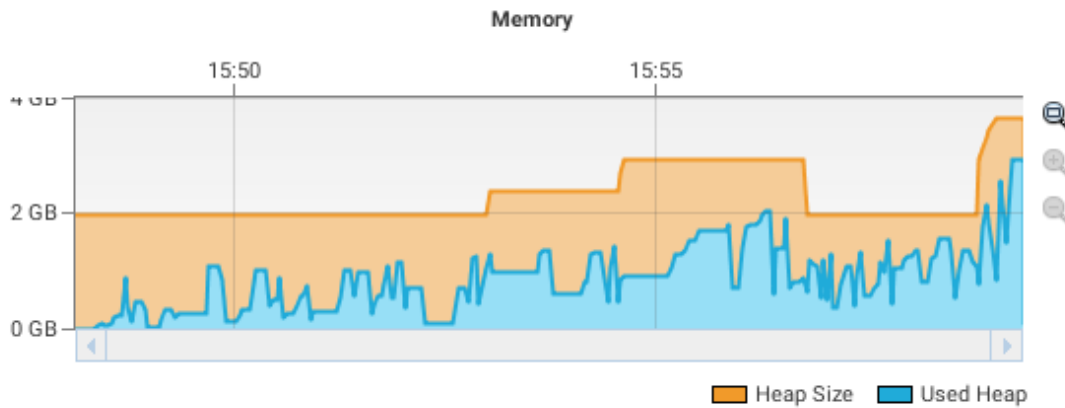


Figure 3.9: Memory consumption after memory optimization

3.5.2 Results

In experiments (2 iterations with default PIT selector, using `MethodAdd:AllLiteralAmplifiers:MethodGeneratorAmplifier` amplifiers, over the entire Dhell test suite), this optimization showed a significant qualitative improvement of memory consumption (see Figure 3.8 and Figure 3.9), particularly in the second half of the DSpot experiment timeframe.

3.6 DSpot Optimization: overall experimentation

In order to evaluate the overall impact of these optimization in DSpot, we have conducted a number of experiments that aims to provide a qualitative assessment. A more precise quantitative assessment of the optimization process is not possible because during this time frame new features of DSpot (which could have an impact on the overall DSpot performance) were developed in parallel (by other DSpot development members). Hence, both the optimization PRs and these new features were merged into the master branch, so there is no way to compare identical versions of DSpot, without and with

the performance optimization improvements. Nonetheless, in these experiments, we compared the latest DSpot version available just before the first optimization PR was merged, with the latest DSpot version at the time of reporting.

Setup: Laptop Dell Latitude E550. CPU: Intel(R) Core(TM) i5-5300U CPU @ 2.30GHz, 2 cores, 4 execution threads, Memory: 16Gb, SSD: 500Gb

Target project: Dhell

DSpot configuration: -i 3 -test eu.stamp_project.examples.dhell.HelloAppTest -a MethodAdd (3 iterations, target test class: eu.stamp_project.examples.dhell.HelloAppTest (JUnit4), applifiers: MethodAdd)

Additional DSpot configuration for parallel test (only for DSpot after optimization): -execute-test-parallel-with-number-processors 4

Experiments

A- Before optimization

Jacoco selector:

===== REPORT =====

Initial instruction coverage: 462 / 1077

42.90%

Amplification results with 42 amplified tests.

Amplified instruction coverage: 466 / 1077

43.27%

[INFO] 2019-09-02 12:50:48 Main - Amplification succeed.

[INFO] 2019-09-02 12:50:48 Main - Elapsed time 263427 ms

[INFO] 2019-09-02 12:50:48 GlobalReport - DSpot amplified your test suite without errors. (no errors report will be outputted)

Total execution time: 4m 28,835s

PIT selector:

===== REPORT =====

PitMutantScoreSelector:

The original test suite kills 15 mutants

The amplification results with 1 new tests

it kills 2 more mutants

[INFO] 2019-09-02 12:42:53 Main - Amplification succeed.

[INFO] 2019-09-02 12:42:53 Main - Elapsed time 910106 ms

[INFO] 2019-09-02 12:42:53 GlobalReport - DSpot amplified your test suite without errors. (no errors report will be outputted)

Total execution time: 15m 15,598s

B- After optimization

Jacoco selector:

2019-09-02 13:07:29,395 INFO eu.stamp_project.Main - Amplification succeed.

2019-09-02 13:07:29,395 INFO eu.stamp_project.Main - Elapsed time 162627 ms

2019-09-02 13:07:29,397 INFO eu.stamp_project.utils.report.output.selector.TestSelectorReport - Initial instruction coverage: 462 / 1077

42.90%

Amplification results with 92 amplified tests.
 Amplified instruction coverage: 463 / 1077
 42.99%

2019-09-02 13:07:29,398 INFO eu.stamp_project.utils.report.output.selector.TestSelectorReport
 - Writing report in dspot-out/report.txt
 2019-09-02 13:07:29,398 INFO eu.stamp_project.utils.report.error.ErrorReport - DSpot amplified your test suite without errors. (no errors report will be outputted)
 2019-09-02 13:07:29,398 INFO eu.stamp_project.utils.report.output.OutputReport - The amplification ends up with 92 amplified test methods over 1 test classes.
 2019-09-02 13:07:29,398 INFO eu.stamp_project.utils.report.output.OutputReport - Print eu.stamp_project.examples.dhell.HelloAppTest.java with 92 amplified test cases in dspot-out/

Total execution time: 2m 48,014s

PIT selector:

2019-09-02 13:15:11,761 INFO eu.stamp_project.Main - Amplification succeed.
 2019-09-02 13:15:11,762 INFO eu.stamp_project.Main - Elapsed time 367505 ms
 2019-09-02 13:15:11,765 INFO eu.stamp_project.utils.report.output.selector.TestSelectorReport
 - Test class that has been amplified: eu.stamp_project.examples.dhell.HelloAppTest
 The original test suite kills 15 mutants
 The amplification results with 1 new tests
 it kills 2 more mutants

2019-09-02 13:15:11,765 INFO eu.stamp_project.utils.report.output.selector.TestSelectorReport
 - Writing report in dspot-out/report.txt
 2019-09-02 13:15:11,765 WARN eu.stamp_project.utils.report.error.ErrorReport - DSpot encountered some errors during amplification.
 2019-09-02 13:15:11,766 WARN eu.stamp_project.utils.report.error.ErrorReport - DSpot encountered 2 error(s) during amplification.
 2019-09-02 13:15:11,766 INFO eu.stamp_project.utils.report.output.OutputReport - The amplification ends up with 141 amplified test methods over 1 test classes.
 2019-09-02 13:15:11,766 INFO eu.stamp_project.utils.report.output.OutputReport - Print eu.stamp_project.examples.dhell.HelloAppTest.java with 141 amplified test cases in dspot-out/

Total execution time: 6m 13,094s

Note that DSpot output report format before and after optimization are different (even results) since we are comparing two different DSpot releases (not only in performance but in features).

We conducted a few experiments (for each configuration) and obtained qualitative similar results. The total DSpot execution times for one of those experiments are shown in Table 3.1.

Table 3.1: Experiments for DSpot optimization on DHell test suite

	Jacoco Selector	PIT Selector
Before optimization	4m 28,835s	15m 15,598s
After optimization	2m 48,014s	6m 13,094s
Improvement (percentage)	37,5%	59,25%

Summarizing, in these experiments we have obtained a DSpot performance improvement in the

range [37%, 59%] depending on the selector used. However, these experiments target the DHell project, whose tests classes are executed in short time frames (around few milliseconds). Therefore, they do not benefit from parallel test execution. When we introduce artificially some delay in each test execution (for instance using `Thread.sleep()`), we expect that the overall DSpot performance improvement using parallel test execution would be notably higher. However, based on preliminary experiments, the effect of parallel test execution on the performance depends on the chosen selector (e.g. Jacoco or PIT), as shown in Table 3.2.

Table 3.2: Experiments for DSpot optimization on DHell test suite (tests delayed 200ms)

	Jacoco Selector	PIT Selector
Before optimization	31m 33,649s	127m 59,522s
After optimization	14m 3,630s	91m 46,680s
Improvement (percentage)	55,44%	28,29%

According to these results the PIT coverage computation seems to dominate (in terms of time computational consumption) over the test execution, so that the benefit of parallel test execution is lower.

Experiments conducted on industrial use cases could yield more concluding results about the overall benefit of test parallel execution for different selectors.

Chapter 4

Descartes

Descartes can be used to find issues in a test suite. It has been conceived as an extension of PITest, a state of the art mutation testing tool. Unlike PITest, Descartes uses extreme mutations/transformations. Compared to traditional mutation testing, the use of extreme transformation brings two advantages: 1) the reasoning of testing issues is done at the method level, which makes them easier to understand and 2) it creates less program variants/mutants which makes the analysis faster.

4.1 Development progress

From the last deliverable ¹ to the moment in which this document is being written ² two new versions have been released to Maven Central. Current stable version is 1.2.5 and version 1.2.6 is under testing before release. During this period, four pull requests from an external contributor have been merged into the main codebase. These pull requests add fixes and enhancements to the code, a new subcategory for stop methods and a new mutation operator. This operator handles methods returning instances of classes having a constructor without parameters. The operator makes the method return an instance created with this constructor. Another mutation operator was also added following a suggestion from another external contributor ³. In this case, the operator targets methods returning instances of `Optional`. The operator makes the method return an “empty” instance. Further development actions should be directed to solve seven test execution issues detected by XWiki.

4.2 Generating automatic test improvement suggestions

Descartes is able to detect the worst tested methods in a project. When faced with the list of these methods, Developers then need to investigate the interplay between the existing test cases and the application code. They need to understand how to solve the testing issue behind the methods where extreme transformations (extreme mutants) were not detected by the test suite.

Here we propose to complement the results provided by Descartes with an *infection-propagation* analysis whose goal is to automatically generate suggestions for developers to fix the highlighted testing issues. The analysis consist in a three stages process implemented in a tool named RENERI that extends the functionalities of Descartes.

¹<https://github.com/STAMP-project/pitest-descartes/releases/tag/D1.3>

²<https://github.com/STAMP-project/pitest-descartes/releases/tag/D1.4>

³<https://github.com/STAMP-project/pitest-descartes/issues/99>

4.2.1 Overview of the process for test improvement suggestions

The *infection-propagation* analysis takes as input a program P , a test suite T and U , a set of extreme transformations that were not detected by T . U is the output provided by Descartes. For the sake of simplicity, we consider P to be a set of methods, T to be a set of test cases and U a set of pairs (m, m') where $m \in P$ and m' is the method after the application of the mutation.

The global process to understand undetected extreme transformations and to synthesize a test improvement suggestion operates in three main stages. Algorithm 1 outlines the process. Each stage is detailed in the following sections:

1. Infection detection: identify the extreme transformations that infect the local immediate state; (line 1, subsection 4.2.2), that is, identify when m' produces a different program state than m with the execution of T .
2. Propagation detection: discover which infections reach some test case states and (line 2, subsection 4.2.3) that is, discover when the execution of T using m' produces a different program state than m that can be observed in one of the existing test cases.
3. Test improvement suggestion: generate the report by consolidating the information gathered in the two previous stages (line 3, subsection 4.2.4).

The initial two stages include a dynamic analysis of the program. Each stage instruments the elements to be observed and executes the test suite with the original code and the transformed method variant. These executions of the instrumented program record the program and test states to help the suggestion generation. The states are compared to discover the symptom that explains each why each transformation was not detected by the test suite.

Data: P : program under test, T : test suite, U : undetected extreme transformations

```

1 no-infection, infection  $\leftarrow$  find_infections( $P, T, U$ )
2 no-propagation, weak-oracle  $\leftarrow$  find_propagations( $P, T, infection$ )
3 gen_suggestions(no-infection, no-propagation, weak-oracle)

```

Algorithm 1: The three stages process implemented by RENERI

The analysis results in the identification of one symptom for each undetected extreme transformation. The identified symptom shall help to explain why the extreme transformation was not detected by the test suite. We identify the following symptoms:

- **no-infection:** there is no observable difference in the local state of the program after the method invocation when the test case is executed on the original and transformed method.
- **no-propagation:** there is an observable difference in the program state at the transformation point, but there is no observable difference in the state of the test case.
- **weak-oracle:** the program state infection is propagated to the code of a test case but no assertion fails.

These symptoms, as well as intermediate observations of the dynamic program analysis are consolidated in a final report to provide concrete improvement suggestions to the developers.

4.2.2 Infection detection

The goal of the first stage is to determine whether the execution of the tests under an extreme transformation infects the program state. We instrument the original program and the program after the application of the extreme transformation to observe the immediate program state in both situations. If we can observe a difference between the states, then this signals an infection. The main steps of this stage are outlined in algorithm 2

Data: P : program under test, T : test suite, U : set of undetected extreme transformations

Result: *no-infection*: {**no-infection** symptoms}, *infection*: {extreme transformations that produce an infection}

```

1 function find_infections( $P, T, U$ ) does
2   foreach  $(m, m') \in U$  do
3      $T_m \leftarrow \{t \in T : t \text{ executes } m\}$ 
4      $P_i \leftarrow \text{instrument } m \text{ in } P$ 
5      $MS_m \leftarrow \text{observe}(P_i, T_m)$ 
6      $P' \leftarrow \text{replace } m \text{ by } m' \text{ in } P$ 
7      $P'_i \leftarrow \text{instrument } m' \text{ in } P'$ 
8      $MS_{m'} \leftarrow \text{observe}(P'_i, T_m)$ 
9      $diff \leftarrow \text{get\_diff}(MS_m, MS_{m'})$ 
10    if  $diff \neq \emptyset$  then
11       $infection \leftarrow infection \cup \{(m, m'), diff\}$ 
12    else
13       $no\_infection \leftarrow no\_infection \cup \{(m, m')\}$ 
14    end
15  end
16  return  $no\_infection, infection$ 
17 end

```

Algorithm 2: Infection detection stage of the process. This stage identifies the extreme transformations that are not detected because of a **no-infection** symptom

```

1 function observe( $P, T$ ) does
2   for  $i \leftarrow 1$  to  $N$  do
3      $S_i \leftarrow \text{execute } T \text{ with } P$ 
4   end
5   return  $\cup_{i=1}^N S_i$ 
6 end

7 function get_diff( $S_1, S_2$ ) does
8    $diff \leftarrow \emptyset$ 
9   foreach  $(p, v) \in S_1$  do
10    if  $\exists (p, w) \in S_2 \wedge v \neq w$  then
11       $diff \leftarrow diff \cup \{(p, v, w)\}$ 
12    end
13  end
14  return  $diff$ 
15 end

```

Algorithm 3: Two functions to compute the state invariants across test executions and find a difference between the state of the original program and the transformed variant.

Instrumentation

For each method and its transformed variants we observe the following parts of the program state: the instance on which the method is invoked, all arguments (if any), and the result value (if the method returns a value).

For primitive values and strings, the observation is trivial. For objects of other types, an observation means inspecting public and private fields using reflection. In this way the observation minimizes potential side-effects to the object being observed.

In practice, we obtain these observations by instrumenting the code of the original methods and its transformed variant (lines 4 and 7 from algorithm 2). The instrumentation targets the compiled bytecode of the program.

Listing 4.1 shows an example of how a is instrumented for observation. The `observe` function saves the state of the object.

```

1 public boolean equals(Object other) {
2     boolean result = ... /* original method code */
3     /* Observation */
4     observe(this);
5     observe(other);
6     observe(result);
7     return result;
8 }

```

Listing 4.1: Method instrumented to observe the immediate program state

Dynamic Analysis

Each extreme transformation $(m, m') \in U$ is observed and analyzed independently. After the instrumentation, all the test cases known to cover m are executed. With this, we observe the original program state MS_m . The same is done with m' to obtain $MS_{m'}$. If there is at least a property with a different value in both states, then the states are said to be different (`get_diff` in algorithm 3). The property points to the code location where the infection has manifested: one of the arguments, the receiver or the result of an invocation.

If no difference is observed, then the process has discovered a **no-infection** symptom for the given extreme transformation.

A method could be invoked several times by the same test case. In our experience, we have seen methods invoked thousands of times in a single test execution. During the dynamic analysis of this stage, each method invocation is uniquely identified by its order in the entire method invocation sequence. This identification differentiates the states from each invocation of m and m' . The properties from one method invocation are distinguished from the properties of another method invocation. For example, the result value of the first invocation of m is considered as a distinct property from the result value of the second invocation of m . With this, the state elements from the first invocation of m are only compared to the same state elements from the first invocation of m' and so on.

During the observations, there may be natural changes from one execution to the other, that are not due to the extreme transformations [1]. Such variations may be derived from random number generation, usage of system time, usage of temporary file paths, and they should be discarded. As a trivial example, in line 9 of Listing 4.2, the value of `start` depends on the current system time. In order to identify such naturally changing values, we execute the tests N times when recording the states of the original methods and another N times when recorded their transformed variants. N is a parameter that can be controlled by the user with a default value of 10.

```

1 @Test
2 public void handlesManyChildren() {
3     // Arrange
4     StringBuilder longBody = new StringBuilder(500000);

```

```

5  for (int i = 0; i < 25000; i++) {
6      longBody.append(i).append("<br>");
7  }
8  // Act
9  long start = System.currentTimeMillis();
10 Document doc = Parser.parseBodyFragment(longBody.toString(), "");
11 // Assert
12 assertEquals(50000, doc.body().childNodesSize());
13 assertTrue(System.currentTimeMillis() - start < 1000);
14 }

```

Listing 4.2: Example of a test case using the system time. The value of `start` changes on every tests execution

For the state comparison, we keep only the properties that had the same value across all N executions. So, we actually consider as the state of the program only the set of state elements that remained unchanged across executions, as can be seen in the `get_diff` function from algorithm 3.

If a difference is observed, we can collect the state property, that is, whether the infection can be observed in a field or the value of the result of the method, in one of the arguments or the receiver.

After this stage, the undetected transformations can be partitioned in two groups: those exhibiting a **no-infection** symptom, and those for which the test executions produce the infection of the immediate program state.

4.2.3 Propagation detection

Data: $P, T, infection$

Result: *no-propagation, weak-oracle*

```

1  function find_propagations( $P, T, infection$ ) does
2      foreach ( $m, m', diff_m$ )  $\in$  infection do
3           $T_m \leftarrow \{ t \in T : t \text{ executes } m \}$ 
4           $T_{m,i} \leftarrow \text{instrument } T_m$ 
5           $TS_m \leftarrow \text{observe}(P, T_{m,i})$ 
6           $P' \leftarrow \text{replace } m \text{ by } m' \text{ in } P$ 
7           $TS_{m'} \leftarrow \text{observe}(P', T_{m,i})$ 
8           $diff_t \leftarrow \text{get\_diff}(TS_m, TS_{m'})$ 
9          if  $diff_t \neq \emptyset$  then
10             | weak-oracle  $\leftarrow$  weak-oracle  $\cup \{(m, m'), diff_t\}$ 
11          else
12             | no-propagation  $\leftarrow$  no-propagation  $\cup \{(m, m'), diff_m\}$ 
13          end
14      end
15      return no-propagation, weak-oracle
16 end

```

Algorithm 4: Second stage of the process. This stage identifies **no-propagation** and **weak-oracle** symptoms

The goal of this second stage is to determine which program state infections, detected in the previous stage, are propagated to the test code. In this stage we observe the state of each test case covering these transformations. At the end, we are able to detect whether the infection could be observed in the existing test code (**weak-oracle**) or not (**no-propagation**). The main steps of this stage are outlined in algorithm 4.

Instrumentation

In this stage, the test states are also observed through source code instrumentation. Unlike in the previous stage, here we instrument the Java source code of the test cases. We instrument only the tests that cover the method of at least one extreme transformation for which a program infection was previously detected. The code is instrumented to observe the state of the values produced by all expressions and subexpressions in the code of the test case. This instrumentation amplifies the observation capabilities of the test cases, limited before only to the values being asserted by the developers.

Listing 4.3 shows a test case and how it is instrumented for observation.

We exclude from the observation: constant expressions and expressions on the left side of an assignment. Generic test method and test classes, that is with type arguments, are not considered in our current implementation. The instrumentation inserts `try...catch` blocks in the test code, in order to observe exceptions that could be thrown during test execution, as it is done in line 24.

```

1 // Original test case
2 @Test
3 public void testAdd() {
4     VersionedSet list = new VersionedSet();
5     list.add(1);
6     assertEquals(1, list.size());
7 }
8
9 // Instrumented test case
10 @Test
11 public void testAdd() throws Exception {
12     try {
13         VersionedSet list =
14             observe(new VersionedSet());
15         observe(list).add(1); // Observes the state of list
16         assertEquals(1,
17             observe( // Observes the result of size
18                 observe(list) // Observes the state of list
19                     .size()
20             ));
21     }
22     catch (Exception exc) {
23         observe(exc);
24         throw exc;
25     }
26 }
27

```

Listing 4.3: Test case instrumented to observe infection propagation

Dynamic analysis

The dynamic analysis in this stage is very similar to the infection detection. Each extreme transformation is analyzed in isolation. The test are executed N times for the original code and N times for the transformed code. Only the state elements that remained unchanged are kept for the state comparison. Unlike the previous stage, here the states are recorded from the test code and the properties point to code locations where developers can actually insert new assertions. If no difference was observed between the execution of the original method and the transformed method, then a **no-propagation** symptom is signaled. The previously detected infection did not propagate to an observable point in the test code. On the contrary, if a difference is observed, then we signal a **weak-oracle** symptom. The existing assertions do not notice the extreme transformation.

The body of the method `example.VersionedSet.equals(java.lang.Object)` was replaced by `return true;`
yet, `example.VersionedSetTest.testEquals` did not fail.

When the transformed method is executed, there is no difference with the execution using the original source code.

This could mean that the original method always returns the same value. Consider creating a modified variant of the test mentioned above to make the method produce a different value.

Figure 4.1: Suggestion generated for a **no-infection** symptom related to the `equals` method in our example.

4.2.4 Suggestion generation

The third stage is about generating the final suggestions. The suggestions are compiled into a human readable report. The report includes a detailed diagnosis of the symptom for each extreme transformation and an indication of the potential solution strategies. In this section we discuss how these reports are generated, the exact information they include and provide an example for each symptom.

Suggestion for a no-infection symptom

For a given undetected extreme transformation, if no immediate program infection is observed for all tests case, it means that the test input is not able to generate a state difference (**no-infection**). It also means that the method had no observable side-effects over the instance on which it was invoked or the arguments. The result value of the method is the same across all invocations. However, there are test cases that reach the method.

Altering the existing input in these test cases could alter the program state so the method can return a different value or produce a different side effect. So, the suggestion generated in the process tells developers *to create new test cases from the ones covering the method* and alter their input to induce an observable effect.

Figure 4.1 shows an screenshot of a suggestion for a **no-infection** symptom.

Suggestion for a no-propagation symptom

A **no-propagation** symptom unveils an immediate program infection that is not propagated to the test code. A different program state is observed after one or more invocations of the transformed method. However, no state difference is observed from the test cases. The effects of the program infection are masked at some point in the execution path. Observing the changed state requires a new invocation sequence able to propagate the infection to an observable point from the test. This means that *new test cases are then required to detect these extreme transformations*. This the suggestion that we generate for these symptoms.

However, the method under study could be private and not directly accessible from the test cases. To address this problem, we find methods that are accessible from the test code, and that are as close as possible in an invocation sequence to the method we want to target. For this we perform a static analysis in the code of the project that follows the invocation graph. The analysis produces the list of public or protected methods inside accessible classes that can be use to reach the method.

The report for a **no-propagation** symptom includes a description of the transformation and the list of methods that should be targeted in the new test cases. If the method is already accessible then it is the only one listed as target. If it is not accessible, we include the list produced by the static analysis. The report also includes the test cases executing the method. Figure 4.2 shows an screenshot of the generated report for a **no-propagation** symptom.

The body of the method `example.VersionedSet.isEmpty()` was replaced by `return true;` yet, `example.VersionedSetTest.testIntersection` did not fail.

It is possible to observe a difference between the program state when the transformed method is executed and the program state when the original method is executed. This difference is observed right after the method invocation but not from the top level code of any test.

For one invocation of `isEmpty`, it was observed that the return value was `true` but should have been `false`.

To solve this problem you may consider to:

- Create a new test case that targets the result of `isEmpty` directly, since it could be accessed from a test class.
- Refactor the code that uses this method. Maybe the method is not actually needed in the context that it is being used.

Figure 4.2: Suggestion generated for a **no-propagation** symptom.

The body of the method `example.VersionedSet.incrementVersion()` was removed yet, none of the following tests failed:

- `example.VersionedSetTest.testIntersection`
- `example.VersionedSetTest.testAdd`

It is possible to observe a difference between the program state when the transformed method is executed and the program state when the original method is executed. This difference can be observed in `VersionedSetTest.java` from the expression returning a value of type `example.VersionedSet` located in line 22 from column 31 to column 34

When the transformation is applied to the method, it was observed that the field `version` of the value obtained from the expression was `0` but should have been `1`.

Consider modifying the test to verify the value of `version` in the result of the expression.

Consider verifying the result or side effects of one of the following methods invoked for the result of the expression:

- `example.VersionedSet.getVersion()`

Figure 4.3: Suggestion generated for a **weak-oracle** symptom.

Suggestion for a weak-oracle symptom

If an infection propagation is detected (**weak-oracle**), it is visible in the result of an expression in the test code. In the previous stage we have recorded the exact code location of this expression.

The **weak-oracle** suggestion is *to add an assertion at the right location in the test code*. We even provide the developer with the value to be asserted.

The state difference could be the result of the expression itself if it is a primitive type or a string. But, if the expression returns an object, the difference may be observed in one of its fields. If the state difference is observed through an accessible field of the result of the expression or the result itself, the suggestion is to create an assertion targeting this value.

If the field is not accessible, (i.e. it is private) further actions are required. For this, we perform a static analysis to find methods that use the identified field. These methods could be not accessible from the test code. So, we find accessible class members invoking these methods, in the same way it is done for the **no-propagation** reports. The final suggestion is to assert the result and side effects of the final set of methods, invoked in the result of the initially identified expression. Figure 4.3 shows an screenshot of the report generated for a **weak-oracle** symptom.

Table 4.1: Projects involved in the empirical validation with developers

Project	Issues	ni	np	wo
Funcon4J	5	2	3	0
greycat	6	2	2	2
sat4j-core	6	2	2	2
xwiki-commons-job	6	2	3	1
Total	23	8	10	5

4.2.5 Implementation

RENTERI has been conceived as a Maven plugin and it is able to target Java programs that use Maven as main build system. The code, and all data related to the current work are available from Github ⁴. RENTERI relies on Javassist 3.24.1 [?] for bytecode instrumentation, Spoon 7.1.0 [?] for static code analysis and Java source code transformation and Descartes 1.2.5 to apply extreme transformations. All stages in the process described before are exposed as Maven goals. Apart from human readable reports, RENTERI also generate files more suitable for automatic analysis that could be used by external tools.

4.2.6 Evaluation

We performed a qualitative evaluation of the suggestions generated by RENTERI. In particular, we want to know if developers find the generated suggestions helpful and if they actually accept the proposed solutions.

We used the four projects listed in Table 4.1. For each project we selected up to six undetected extreme transformations for which RENTERI generated a suggestion. These transformations were manually selected by us so that they represent interesting testing cases for the developers. Table 4.1 shows the projects and the symptoms of the selected issues.

For each issue we created a form containing a link to the automatically generated report (similar to those shown in Figures 4.1, 4.2 and 4.3); and a set of questions, as shown in Appendix 4.3.

Qualitative feedback from the developers

Table 4.2 summarizes the feedback given by the developers we consulted. It contains the answers from the developers divided according to the three types of symptoms. As can be seen, of the 23 issues that were discussed, 18 were considered as relevant or of medium relevance. The developers considered the report to contain helpful information in 18 cases, and even thought that the exact testing solution was given in 4 cases. In all cases, the developers, could emit a verdict about all the testing issues in around 30 minutes.

In three cases, two **no-propagation** and one **no-infection**, the developers found that the synthesized suggestion was not helpful. In two of those cases, the developers argued that the suggestion should contain more information about the state differences between the executions of the original and the transformed method. For example, developers would like the tool to identify the instruction in the test code that triggered the method invocation for which the state difference was observed. The third case was a method related to the performance of the code. The developer argued that a test case for this method would be “artificial”.

On the other hand, two suggestions were considered as misleading. In one case, the issue could not be reproduced by the developer. In the other, the developer could not make the connection between the program state difference and the test cases executing the method.

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

Table 4.2: Summary of the feedback given by developers

	ni	np	wo	Total
The issue in the description is:				
relevant	4	5	2	11
of medium relevance	2	4	1	7
not important	1	1	1	3
not really a testing issue		1	1	2
The suggestion provided:				
points to the exact solution	3		1	4
provides helpful information	4	7	3	14
is not really helpful	1	2		3
is misleading		1	1	2
Developers solve the issue by:				
adding a new assertion	1	4	1	6
slightly modifying a test case	1			1
creating a new test case	3	1	1	5
performing other actions	3	5	3	11

So, developers find the suggestions not helpful or misleading when they fail to understand what caused the program state difference between the executions of the original and the transformed method. Suggestions could be further improved with additional information, for example, the stack trace containing the invocation sequence from the test code to method in **no-propagation** symptoms.

Developers considered that six transformations can be solved with the addition of an assertion, only one by slightly modifying an existing test case and five with the creation of a new test case. In 11 cases developers considered that the transformations should be solved by other actions. These actions included, for example, a complete replacement of the assertions in a test case, or a combination of new input and new assertions, or, instead of adding a new verification to an existing test case, developers would prefer to create a new test case by repeating the same code with the new assertion. The solutions are in fact influenced by the testing practices of each developer.

Even when developers would not modify the test code as suggested, the report provides information that it is generally considered as helpful and the proposed solution as a valid starting point to ease the understanding of the testing issue.

Actual solutions given by the developers

Developers actually solved 13 issues with 11 commits. Table 4.3 shows the links and projects for all commits. One issue in `funcon4j` was not solved as it was related to a testability problem. No issue was fixed for `greycat` due to non-technical and unrelated reasons. In this section we provide two examples of how developers solved the extreme transformations and how do the solutions relate to the synthesized suggestion.

In project `funcon4j` the developer was presented with a **no-infection** symptom. The suggestion, was to create a modified variant of the existing test case to make the method produce a different value. In the questionnaire, the developer answered that the issue can be solved with the addition of a new assertions. The actual action taken by the developer is shown in Listing 4.4, that is, the addition of line 2. The actual commit can be consulted in <https://github.com/manuelleduc/Funcon4J/commit/63722262313fb2dac5b516bbae5f04e0502e7f26>.

As we can see, the developer added a new assertion but she also included a new input derived from

Table 4.3: Commits created by developers with the help of RENERI

Project	Commit
funcon4j	63722262313fb2dac5b516bbae5f04e0502e7f26
funcon4j	497574909329abcaa6987a9b3588ef3da0aebf44
funcon4j	bea72c8ddcf0a848dfe455a0280184647e3b3d2a
funcon4j	ba746f59818119c9abf61396e9c2ecd40f0e8f28
sat4j-core	d3e769790700806107ebcc1766f000a1be216460
sat4j-core	a2de38673f8b727875cf30ed791e59f9ee1a7a80
xwiki-commons-job	3c851e4d0255b8ab1ab4610a289638cbc0051d58
xwiki-commons-job	2cdd7dc94eafd31de0ed7888ba2602702fa111b9
xwiki-commons-job	77d453048f2b799625837942864df07b7417c7d7
xwiki-commons-job	da7ef4a4da42b8f5771d99c030d14b3b1066be84
xwiki-commons-job	b6b97a8f8f0a707af775d410f1d5af13dc5a1d16

the existing code, using a small modification. So in fact, the actual testing solution is in line with the report generated by RENERI.

```

1 test("{a = 1, b = 2} > {b = 1, a = 1};;", "true");
2 test("{a = 1, b = 1} > {b = 1, a = 1};;", "false"); // Fixture
3 test("\abc\" > \"abd\";;", "false");

```

Listing 4.4: Example of a test fixture created by a developer based on the information contained in the RENERI report

In Listing 4.5 we show an example of a test code fixture created by one of the developers of project `sat4j-core` with the help of RENERI. The original test code is shown between lines 2 and 8. The **report** pointed at line 6. Under the extreme transformation, the method invocation produced a null value while the original code should produce a non-null array. The new test fixture can be seen between lines 11 and 21. The developer confirmed that RENERI provided the exact solution and added the proposed assertion (see line 18). Interestingly enough, an extra assertion was also added in the following line.

```

1 // Original code
2 clause.push(-3);
3 solver.addClause(clause);
4 int counter = 0;
5 while (solver.isSatisfiable() && counter < 10) {
6     solver.model();
7     counter++;
8 }
9
10 // Fixture created by the developer
11 clause.push(-3);
12 solver.addClause(clause);
13 int counter = 0;
14 int[] model;
15 while (solver.isSatisfiable() && counter < 10) {
16     solver.model();
17     model = solver.model();
18     assertNotNull(model); //Fixture
19     assertEquals(3, model.length); //Fixture
20     counter++;
21 }

```

Listing 4.5: Example of the addition of an assertion guided by the generated report

So, we conclude that according to our empirical evaluation, the information in the synthesized suggestions is helpful for developers. In the best case, RENERI even points to the exact solution. The

cases where the suggestion was misleading or not helpful may be considered by future research, with more information from dynamic analysis. The real test fixtures created by the developers to fix the testing problem are in line with the suggestions synthesized by RENERI.

4.3 Appendix

Questions included on the form describing each testing issue and its corresponding hint. All the options inside a question are exclusive.

Please check the testing issue and hint described in the following URL <https://github.com/...>

- The issue in the description is:
 - relevant
 - of medium relevance
 - not important
 - not really a testing issue
- You would solve the issue by:
 - adding a new assertion to an existing test case (Possibly adding a new method call as well)
 - creating a new test case which is a slight modification of an existing test case
 - creating a completely new test case
 - Other (Specify)
- If you actually solve the issue, please share the URL to the commit
- The suggestion provided in the description
 - points to the exact solution
 - provides helpful information to solve the issue
 - is not really helpful
 - is misleading
- Please, include here any general feedback you would like to provide regarding this issue

Chapter 5

Industrialization

The big challenge when new tools are introduced within an industrial software development toolchain is that they should be introduced seamlessly, ideally with no or minimal change or impact in the code base and in developer daily activities. This becomes particularly true when DevOps principles are applied, and continuous integration and continuous delivery processes are put in place. According to excellent book "Leading the Transformation" from Gary Gruver and Tommy Mouser, applying DevOps principles at scale for a business has unique challenges that are best addressed through Continuous Delivery: it automates as much of the configuration, deployment, and testing processes as possible, and this is done to ensure the repeatability. The automation helps to eliminate human errors, improves consistency, and supports increased frequency of deployments. But, as pointed out in another excellent book, "Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation" from Jez Humble and David Farley, initially these processes are very painful, and there are several issues to fix in order to make everything to work as well as possible.

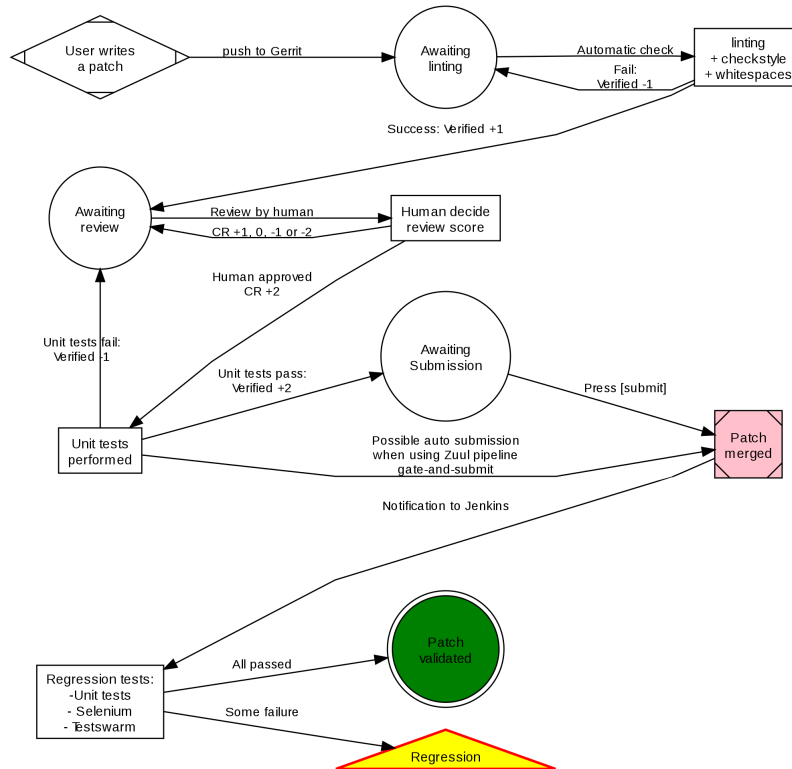


Figure 5.1: Continuous integration workflow (source: Hashar (https://commons.wikimedia.org/wiki/File:Wikimedia_CI_workflow.svg), "Wikimedia CI workflow", <https://creativecommons.org/licenses/by-sa/3.0/legalcode>)

Once continuous integration and continuous delivery processes are fine tuned, every new change needs to be carefully evaluated, and every new tool adoption needs to be as seamless as possible.

At the core of continuous integration/continuous delivery there is an Orchestrator, a tool that coordinates all of this automation. It enables you to call scripts to create new environments on a set of virtual machines or containers with scripted environments, deploy code to that environment using scripted deployment, and then kick off automated testing. At the end of the automated testing, if everything passes, it can move this version of the applications and scripts forward to the next stage of the delivery pipeline.

Industrialization process of STAMP unit test amplification tools aimed to provide means to seamlessly embed test assessment (provided by Descartes) and test amplification (provided by DSpot) within continuous integration/continuous delivery pipelines. According to this goal we focused on the following three key points:

- identify a proper CI/CD orchestration solution in order to make STAMP adoption like a breeze as much as possible
- produce the building blocks to enable test amplification within the orchestrator
- investigating the best approach to use those building blocks

5.1 CI/CD solution selection

The most natural choice for a CI/CD solution is Jenkins: it is an open source automation tool written in Java, with a rich ecosystem of plugins, to achieve Continuous Integration, Continuous Delivery and

Continuous Deployment adoption within organizations, thanks to a well defined model to configure processes to automate as much as possible the entire SLDC:

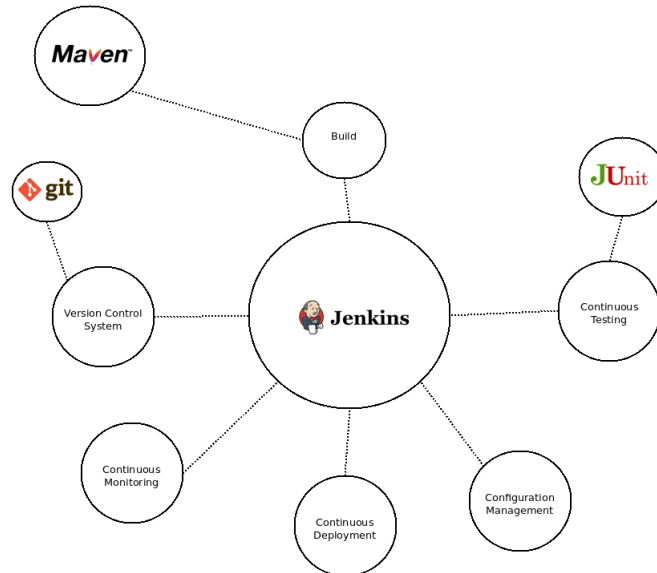


Figure 5.2: Continuous integration/Continuous Deliver orchestrated by Jenkins

In this schema we highlighted the elements which characterize the continuous integration phase, emphasizing the test automation. Gary Gruver and Tommy Mouser ([4]) define Continuous Integration as "a process that monitors the source code management tool for any changes and then automatically triggers a build and the associated automated build acceptance testing. This is required to ensure the code maintains a certain base level of stability and all changes are being integrated often to find conflicts early." Of course, code stability is granted by a strong test base, and here come Descartes and DSpot to enhance testing capabilities of a CI infrastructure. One step ahead to better integrate STAMP unit testing amplification in Continuous Integration has been the choice to adopt Blue Ocean Jenkins extension, a project which rethinks the user experience of Jenkins: information which really matters to development teams are highlighted, while keeping in place the extensibility which is at the core of Jenkins:

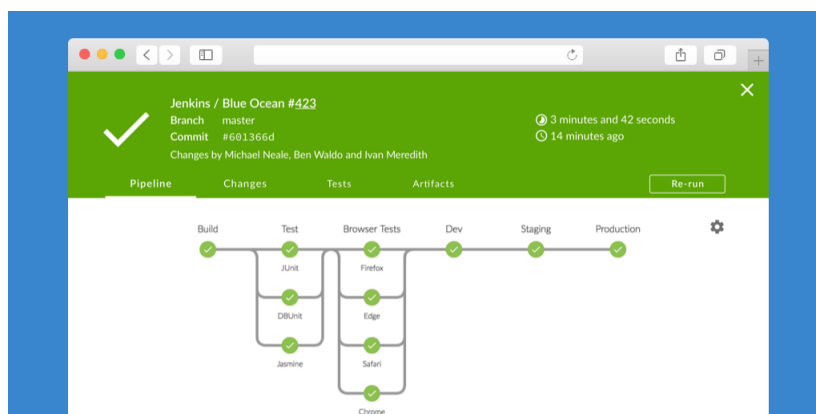


Figure 5.3: Blue Ocean example

5.2 Building blocks

When building software projects, developers need to perform several tasks irrespective of the development methodology used: collecting requirements, designing the solution, developing the code, testing it, etc. These tasks need to be performed in that order, and Maven is some kind of assistant which helps developers in making these things in the right order, in a repeatable way, downloading required third party libraries (also known as JARs) and their dependencies too, build all components and create a deployable package. Furthermore, Maven can be empowered with new skills as the journey proceeds. For these reasons we decided to focus on Maven, in order to make it the perfect assistant to introduce in a standardized way the test amplification concepts in development processes (traditional or agile, it doesn't matter). So STAMP features has been eventually exposed as ordinary Maven goals. This activity led to have a Maven plugin which exposes DSpot amplification features as an ordinary Maven goal (<https://github.com/STAMP-project/dspot/dspot-maven>), which can be in turn applied also to multi-module Maven projects. Descartes, already developed as a Pit plugin, became the perfect companion for DSpot test amplification capabilities, providing him with another test assesment tool more effective than Pit itself. The development of PitMP completed the big picture, in order to have extreme mutation capabilities available also for Maven multi-module projects. These plugins evolved along with new versions of the tools, and provided feedback to optimize several aspects (how to pass tool parameters and so on).

5.3 Industrial adoption

The last project phase focused on defining the adoption for DSpot and Descartes/PitMP in a industrialized context, making them as ordinary tools available to all developers, and at the same time promoting the contribution from new developers to DSpot and Descartes evolution, defining standard project structures.

5.3.1 DSpot and Descartes execution in Blue Ocean

The entire test assessment/test amplification process has been automated thanks to the development of specific pipelines able to perform following tasks:

1. when a new code change is pushed back on repository, make an assessment of existing test cases
2. then amplify existing test cases and push them in a dedicated branch
3. then open a pull request with amplified test cases
4. when new test cases are available in a new branch asses them

Actually more combination of test assessment/test amplification are possible, it is simply a matter of configuring a Jenkins pipeline. Usage of Blue Ocean interface makes clear the entire process:

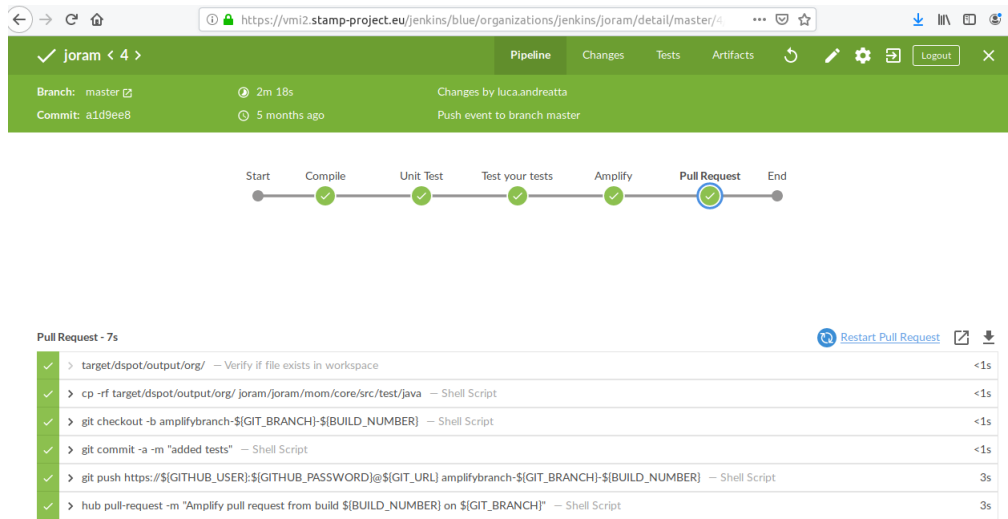


Figure 5.4: Test amplification as seen in Blue Ocean user interface

In this case the entire process of amplifying test cases and then opening a pull request is quite clear: Jenkins detects a code change, so triggers a new build, executes unit test, then amplifies them, opening a new branch and pushing them in it.

This new push event is detected by Jenkins, which in turns simply assesses the new test cases (to avoid infinite loop of test amplification):

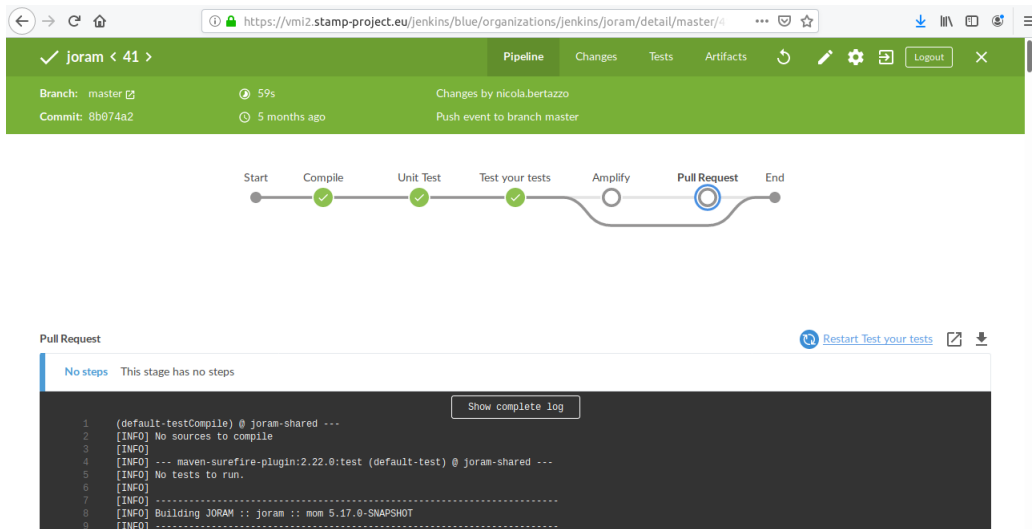


Figure 5.5: Test assessment as seen in Blue Ocean user interface

In this case developers can evaluate through Jenkins interface the quality of new test cases and decide to merge them in the code base.

5.3.2 Descartes/PitMP integration test

Integration and functional tests have a twofold goal: verifying and validating software on one side, and providing developers about technical information on how to use it and how to integrate it. Early

releases of PitMP didn't have structured test suites, validation and verification have been achieving with several shell scripts which tested it. The following picture shows the old PitMP project structure:

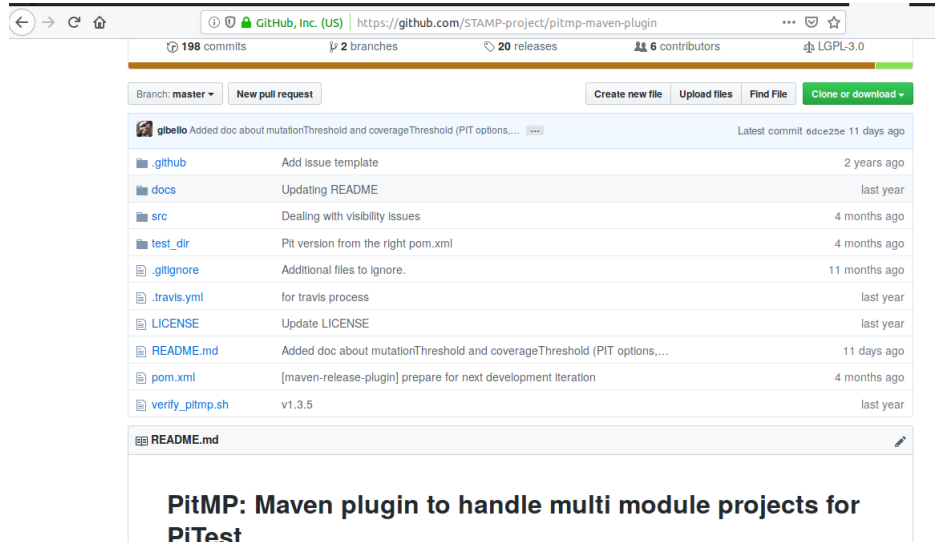


Figure 5.6: PitMP old project structure

In particular the shell script `verify_pitmp.sh` was delegated to test PitMP Maven plugin: it worked well in verifying PitMP new releases, but it made more difficult for external developers to figure out how PitMP was built and eventually to step in the project and contribute to it.

So, given that Maven itself uses `maven-verifier` to run its core integration tests, we developed integration and functional test with `maven-verifier`. These tests are run using JUnit or TestNG, and provide a simple class allowing you to launch Maven and assert on its log file and built artifacts. It also provides a `ResourceExtractor`, which extracts a Maven project from your `src/test/resources` directory into a temporary working directory to let you perform more complex verification and validation activities.

Moreover we organized the PitMP code base giving it the same layout of original Pit project, in order to ease and speed up the process of making PitMP part of Pit ecosystem: PitMP is the missing element in order to have mutation testing also in Maven multi-module projects, and in turn to have Descartes extreme mutation testing in Maven multi-module projects.

Following picture shows how Pit project layout is organized:

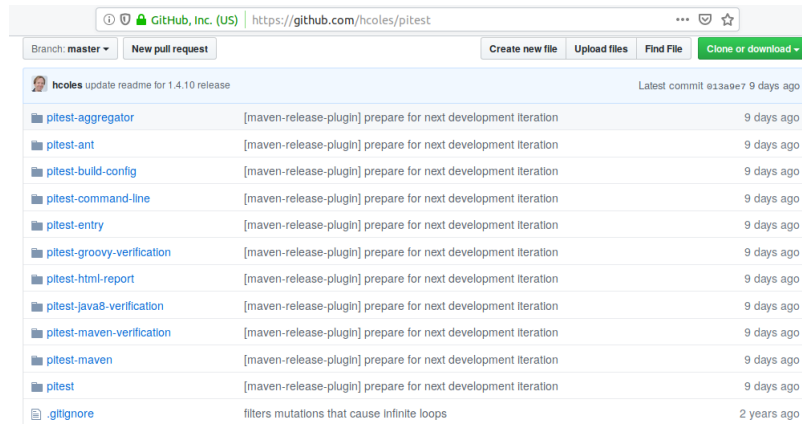


Figure 5.7: Pit Maven project

The `pitest-maven-verification` folder contains Pit integration and functional tests. The same approach holds for PitMP; the new structure is showed in the following picture:

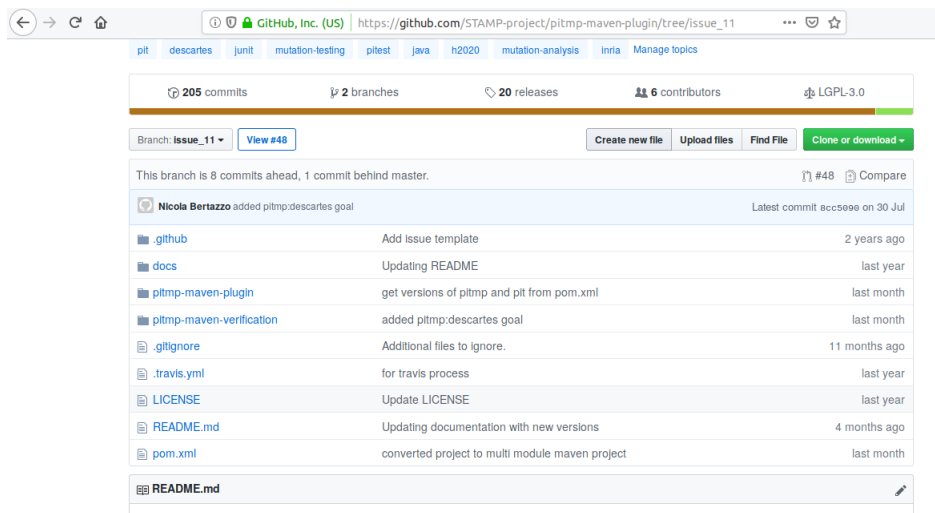


Figure 5.8: PitMP new project structure

Having the code base organized in this standard way let easier contribute to PitMP from STAMP developer community.

5.3.3 Embedding DSpot and Descartes in Jenkins

While the development of DSpot and Descartes Maven plugin opened a way to integrate test assessment and test amplification within Jenkins pipelines, we also developed two Jenkins plugins able to visualize Dspot and Descartes specific reports. These plugins can also combined with declarative Jenkins pipelines, adding an optional step to visualize the report within Jenkins dashboard. Descartes Jenkins plugin was developed during 2018, and in following months we performed several maintenance activities (bug fixing, updating with new versions of Descartes). DSpot Jenkins plugin development started at the end of 2018, and, as Descartes plugin, it provides a dashboard to inspect

and possibly download amplified test cases. The following picture shows the overview dashboard provided by the plugin within each Jenkins build:

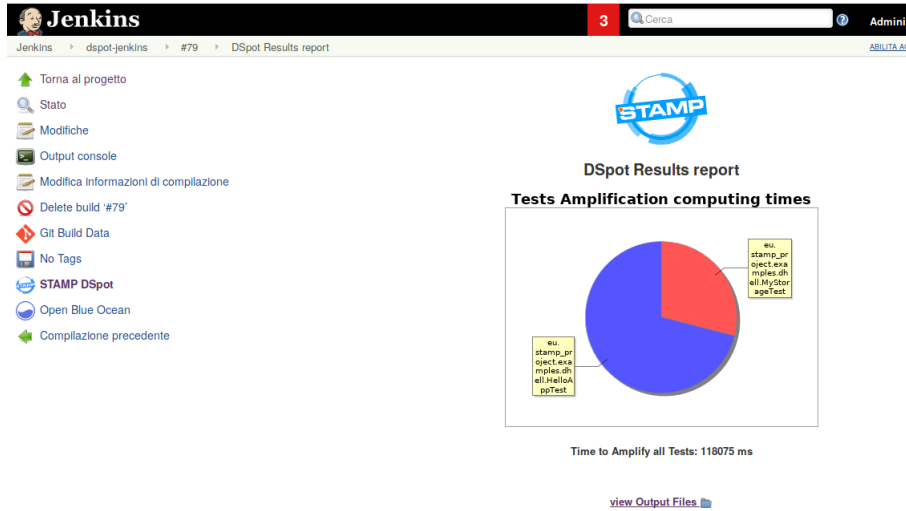


Figure 5.9: Overview of test amplification showed by Jenkins DSpot plugin

The pie chart shows the distribution of time needed to amplify existing test cases. Below the pie chart, there is an overview of amplified test cases, and it is possible to drill down to amplification results and amplified test cases details:

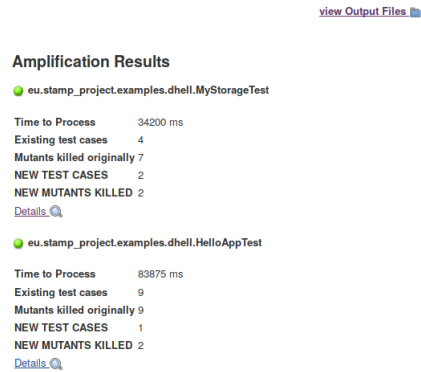


Figure 5.10: Overview on amplified test cases

Developer can inspect directly the amplification result within the current build workspace:



Figure 5.11: DSpot results in current build workspace

Developer can eventually drill down amplification details provided by DSpot:

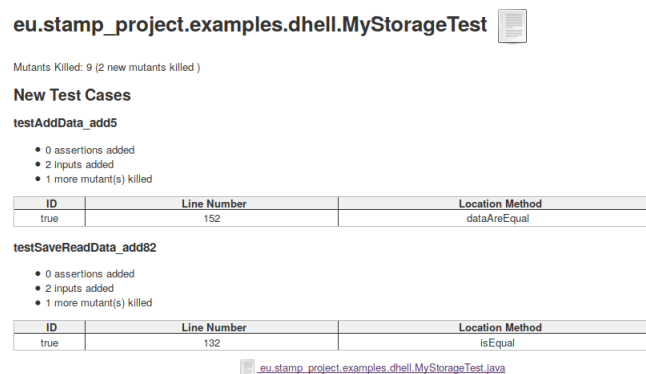


Figure 5.12: Single test case amplification detail

5.3.4 DSpot execution optimization in CI

One of the the goals of the CI integration development is optimizing execution time of DSpot in the CI/CD (for every commit on a branch), executing DSpot just on existing test cases related to code changes, and on new test cases.

Below the description of a typical scenario:

1. From developer workstation:
 - (a) Push an initial version (v1) into the repository
 - (b) Push the new version (v2) into the repository
2. In CI/CD Server (on second push event):
 - (a) checkout the new version (v2)
 - (b) find and download the last stable version that passed the build in the past (v1)
 - (c) Compare the two versions of the project (v2 and v1) to find tests to amplify with DSpot:
 - i. the old tests in v1 that are impacted from the modifications present in production code in v2
 - ii. new tests introduced in v2

In order to understand which is the best approach to detect only a subset of test cases to amplify, we conducted a comparative experiment, using two distinct tools:

1. DSpot Diff tool, used in conjunction with DSpot

2. Jenkins Changeset

As documented in the repo, the main goal of this tool is: "Diff-Test-Selection aims at selecting the subset of test classes and methods that execute the changed code between two versions of the same program". Dspot-diff-test-selector is intended to be used to detect regressions: given a new version of the repo, dspot-diff is able to detect which test cases are related to source code modification. to DSpot which amplifies them, and then executes amplified test cases to the new version in order to detect whether new version passes amplified tests or not. This can be used as a system to evaluate quality of a pull-request before merging it into the master:

- Create a pull request (v2) on the repository
- checkout the code of the master branch (v1)
- checkout the code of the pull request branch (v2)
- from the master branch find only the old test (v1) that are impacted from the changes of v2 (and not the new tests)
- Amplify the old version (v1) to find whether the amplified tests, impacted from the changes of v2, have a regression (pass on v1 and fail in v2) or not

Jenkins changeset feature is widely used in declarative Jenkins pipelines, in order to detect a condition based on specific changes on the source code: Jenkins change-set finds diffs between two different version. It can be used to detect modified test cases, to amplify them, in our context. So we can enrich our toolbox with these tools: use dspot-diff-test selector, if you want to detect a regression, and use Jenkins changeset, in order to amplify only a subset of all test cases (otherwise test amplification could take too much time to execute).

Chapter 6

Conclusion

Over the past year, the work in WP1 spanned tasks 1.2 to 1.5. The development of novel Descartes features to automatically suggest test improvements has been the main activity in T1.2. In T1.3, we have extended DSpot to target a specific commit in the CI, leading to the development of DCI (Detecting behavioral changes in the CI). We have also improved the runtime performance of DSpot, as part of T1.4. Task 1.5 has focused on consolidating the integration of DSpot and Descartes in state of the art DevOps pipelines.

6.1 Publications for WP1

Accepted for publication in August 2019

B. Danglot, O. L. Vera-Pérez, Z. Yu, A. Zaidman, M. Monperrus and B. Baudry. A Snowballing Literature Study on Test Amplification. *Journal of Systems and Software*, 2019.

Major revision submitted in August 2019

B. Danglot, O. L. Vera-Pérez, B. Baudry, and M. Monperrus. Automatic test improvement with dspot: a study with ten mature open-source projects. *Empirical Software Engineering*, 2019. <https://arxiv.org/pdf/1902.08482.pdf>

Submitted in August 2019

O. L. Vera-Pérez, B. Danglot, M. Monperrus, and B. Baudry. Suggestions on Test Suite Improvements with Automatic Infection and Propagation Analysis. *Transactions on Software Engineering*, 2019.

Bibliography

- [1] B. Danglot, O. L. Vera-Pérez, B. Baudry, and M. Monperrus. Automatic test improvement with dspot: a study with ten mature open-source projects. *Empirical Software Engineering*, Apr 2019.
- [2] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. Reassert: Suggesting repairs for broken unit tests. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 433–444, Nov 2009.
- [3] M. Hilton, J. Bell, and D. Marinov. A large-scale study of test coverage evolution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 53–63, New York, NY, USA, 2018. ACM.
- [4] D. F. Jez Humble. *Continuous Delivery*. 2011.
- [5] D. Saff and M. D. Ernst. An experimental evaluation of continuous testing during development. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 76–85. ACM, 2004.
- [6] O. L. Vera-Pérez, B. Danglot, M. Monperrus, and B. Baudry. A comprehensive study of pseudo-tested methods. *Empirical Software Engineering*, Sep 2018.
- [7] J. M. Voas and K. W. Miller. Software testability: The new verification. *IEEE software*, 12(3):17–28, 1995.
- [8] P. Zhang and S. Elbaum. Amplifying tests to validate exception handling code. In *Proc. of Int. Conf. on Software Engineering (ICSE)*, pages 595–605. IEEE Press, 2012.