STAMP

Deliverable D1.3

Enhanced prototype of the unit test amplification tool and report on the performance

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

Deliverable Number	:	D1.3
Title of Deliverable	:	Enhanced prototype of the unit test amplification tool and report on the performance
Dissemination Level	:	Public
Version	:	1.00
Latest version	:	https://github.com/STAMP-project/docs-forum/blob/master/docs/d13_enhanced_prototype_unit_test_amplification_tool.pdf
Contractual Delivery Date	:	M20 July, 31 2018
Contributing WPs	:	WP 1
Editor(s)	:	Benoit Baudry, KTH
Author(s)	:	Benoit Baudry, KTH Benjamin Danglot, INRIA Daniele Gagliardi, Engineering Caroline Landry, INRIA Vincent Massol, XWiki Martin Monperrus, KTH Oscar Luis Vera-Pérez, INRIA Andy Zaidman, TU Delft
Reviewer(s)	:	Andy Zaidman, TU Delft

Abstract

This deliverable reports on the updates towards the construction of a tool-box for unit test 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 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. This tool-box is enhanced with various tools that support the integration of unit test amplification in DevOps pipelines.

KEYWORDS

Keyword List

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

Revision History

Version	Type of Change	Author(s)
1.0	initial setup	Benoit Baudry, KTH
1.1	Descartes chapter	Oscar Luis Vera Perez, INRIA
1.2	flaky tool	Vincent Massol, XWiki
1.3	DSpot	Benjamin Danglot, INRIA
1.4	pitmp	Caroline Landry, INRIA
1.5	integration tools	Daniele Gagliardi, ENG
1.6	edition / review	Andy Zaidman, TU Delft
1.7	edition / review	Benoit Baudry, KTH
1.8	edition / review	Caroline Landry, INRIA

Contents

Chapter 1

Introduction

This deliverable reports on the progress 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 automatic unit test amplification synthesize actionable hints for developers to improve their test suite?

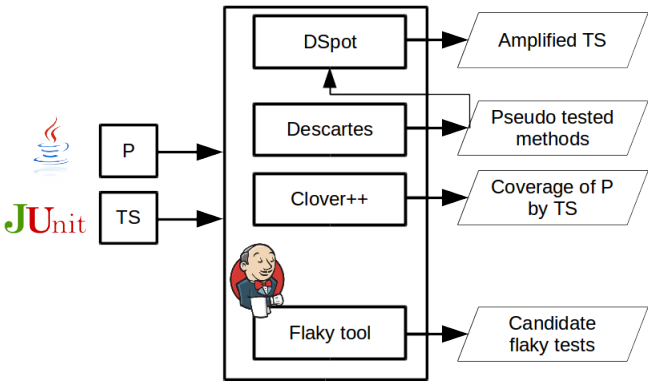


Figure 1.1: Overview of the test amplification tool-box

This deliverable reports on the tool-box that we have built 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, and perform a number of analyses, which provide hints in different forms to the developers:

- edits for existing test cases or proposals for new test cases (as discussed in chapter 2)
- methods that are not well tested, according to extreme mutation (chapter 3)
- test cases that are suspected to be flaky (chapter 4)

The tools and results presented here are follow ups on the tools and results presented 9 months ago in deliverable D1.2. The key novelty lies in a set of integration tools developed to support the integration of our core technical bricks in various DevOps pipelines. These tools are presented in chapter 5.

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 developed a set of tools to explore and measure the complex interplays that exist between a test suite and the program under test: Descartes, a completely novel tool to detect pseudo methods (chapter 3), advanced CI jobs to spot flaky tests and compute global coverage of large projects (chapter 4)
Task 1.3	We build DSpot to investigate the amplification of unit tests (chapter 2)
Task 1.4	Will start after the delivery of this report, per the plan
Task 1.5	We develop a set of tools to integrate amplification in standard DevOps pipelines (chapter 5)

Chapter 2

DSpot: tool for unit test amplification

2.1 Overview

The ultimate goal of DSpot is to automatically synthesize test improvements, i.e., modifications of existing test cases, which are committed to the main test code repository. Our tool takes as input an object-oriented program and its test suite. From the existing tests, it synthesizes test improvements in the form of diffs that are proposed to the developer. For instance, Figure 2.1 shows a test improvement suggested by DSpot. The diffs are meant to be proposed as pull requests in a collaborative development setting such as Github.

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

Figure 2.1: Example of what DSpot produces: a diff to improve an existing test case.

There are two main kinds of improvement that can be done in a test case: assessing the behavior according to a new execution path (for new inputs), and better assessing the correctness of the behaviour after one execution (for existing inputs). In DSpot, both improvements are done: DSpot proposes test modifications that improve existing tests both by triggering new execution paths and by strengthening assertions.

Hence, DSpot's work flow is composed of 2 main phases: 1) transformation of test code to create new test inputs, we call this "input space exploration"; 2) addition of new assertions with oracles to verify the execution state for new test inputs, we call this phase "assertion improvement". In DSpot, "amplification" means the combination of input space exploration and assertion improvement. The following subsections are constructed on those two kinds of exploration.

2.2 Concepts

2.2.1 Definitions

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

Test suite is a set of test classes.

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

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

Unit test is a test method that tests a single unit or component of the system. Typically, unit tests execute a small amount of code.

2.2.2 Test Case Structure

We consider traditional test cases for object-oriented programs. They are typically composed of two parts: input setup and assertions.

The input setup part is responsible for driving the program into a specific state. For instance, one creates objects and invokes methods on them to produce a specific state.

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

2.2.3 Input Space and Observation Space

We define the "input space" as the set of all possible test inputs under which the program can be exercised. This input space is an idealized concept because in practice it cannot be enumerated: the input space is infinite. The "specified input space" consists of points in the input space that are used in the existing test suite. The remaining points are considered as unspecified. DSpot aims at specifying unspecified points.

We define the "observation space" as the set of all possible observation points on a program under test. Technically, an observation point is a call to a public method, that allows to access a value describing one part of the execution state of the program under test.

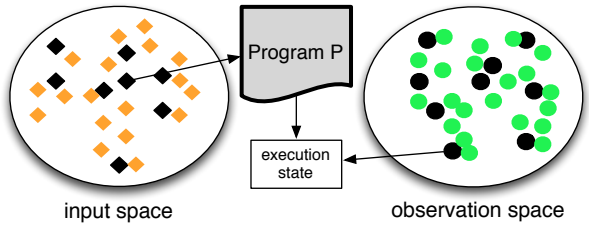


Figure 2.2: On the left the input space is composed by specified input points (orange diamond) and unspecified input points (black diamonds). On the right, the observation space over a given execution state of program. The green circles are values that are already asserted in the existing test suite, while the newly added observations are represented by black circles.

2.3 Improvement of DSpot since D1.2

In this section, we list the new features, bug fixes and general improvement made on DSpot since the last deliverable.

- Amplify specific test methods according to a git diff. This feature is for now in beta version, but strongly needed for the future of DSpot. It will enable to scale-up.

- New test selection strategy. This strategy executes the amplified test methods on another version of the same program and keep the tests that fails. This aims at the detection of regression bugs in the context of continuous integration.
- Match more types of assertions from different testing framework, e.g., google.truth, assertj, etc.
- Generate now assertions on maps, arrays and collections by getting samples of elements.
- Minimizers. Minimizers aims at removing all redundant and useless statements added by amplification, in order to make the amplified test methods clearer.
- A first version of a maven plugin, that allows to launch DSpot from command line
- New STAMP artifact, named testrunner. This tools launches a new JVM and executes the tests. This is done to avoid conflict of dependencies.
- Group id of DSpot, and its plugins is now eu.stamp_project instead of fr.inria.
- Convert automaticcally JUnit3 test methods into JUnit4.
- Refactor Input Amplification to be more unified and extensible.
- Budgetizers. Budgetizers implement a strategy on the Input amplification, avoiding waste of resources. For now there are two Budgetizers: NoBudgetizer and SimpleBudgetizer. The former runs the whole input amplification and reduces the number of amplified test methods according to the diversity of their string representation. The latter takes a fair number between input amplification strategies. For each input amplification strategy, we take the same number of amplified test methods.

2.4 Behavioral changes detection evaluation of DSpot.

In this section, we present a new experiment done using DSpot, in the context of a specific DevOps development process, i.e., pull request based development on GitHub.

2.4.1 Overview

We aim at enhancing the ability of a test suite at detecting behavioral changes between subsequent versions of a program. In the pull request scenario, one developer proposes her changes, but no new unit test methods that characterize it. The goals is to amplify existing test methods to obtain one test method that encodes the changes.

To do this, we use both versions of the same program: the *base*, the version on which the developer wants to merge its changes, and the *Pull request* branch, which contains the changes. DSpot selects the test to be amplified according to the diff between *base* and *Pull request*. Then, it amplifies selected test methods using the *base* as ground truth. And finally, it keeps only amplified test methods that pass on the *base*, but fail on the *Pull request*.

When a test passes on a given version but fails on other, it means that the test method encodes the behavioral difference between both versions.

2.4.2 Pull request based Development & Integration

Our study considers software development processes based on pull requests (PR). On GitHub, a pull request is a way for developers to propose their changes to the community of a given project. The pull request is composed of changes to the code, either source or test code, and has a title and a description. When a developer submits a PR, she creates a branch of the project that contains the proposed change.

We call this branch the *Pull request* branch. The base version in which the developer wishes to merge her PR is called *base* branch.

Before merging the *Pull request* into the *base*, the PR goes through a two-fold review process: 1) The continuous integration process (PR) is triggered. The PR automatically runs the test suites to detect the presence of regression bugs, i.e., behavioral changes introduced by the PR that impact the correctness of the program. However, the PR may fail to spot regression bugs if the test suite is too weak. 2) other developers look at the proposed changes and can either accept them or discuss their substance and form.

Pull request example

Figure 2.3 displays a screenshot of a pull request on *javapoet* [1]:

Figure 2.3: Screenshot of a real pull request on GitHub

We can see the title of the pull request and the diff between the *base* branch and the *Pull request* branch, which is one addition and four deletions to the source code. We see the current state of the pull request, "merged", which means that the changes have been accepted and added to the *base* branch.

Graphical signature of a pull request

We define a graphical signature for a pull request as the column-vector that summarizes the execution of a test suite on two versions of a program. The vector contains markers ✓ and ✗. The ✓ means that the entire test suite passes, and the ✗ means that one or more tests fail. Each row is two version of the same program, *base* and *Pull request* branches.

With a strong test suite, we would have a column-vector identical to the following one

	Status
base	✓
Pull request	✗

A test suite in a pull request should identify behavioral changes. For example, ideally TS would pass on *base* but fail on *Pull request* because it detects and does not accept the behavioral change. The *base* test suite, TS, passes on the *base* branch, but fails on the *Pull request* branch. This means that the test suite specifies enough behavior to detect the difference between the two versions. In this case, *we say the test suite detects a behavioral change.*

[1] https://github.com/square/javapoet/pull/608/files

2.4.3 Behavioral Change

We define behavioral changes as modifications in the source code that imply a new state of the program for the same inputs. Considering the two branches *base* and *Pull request* and the test suite TS that exercises a set of inputs I, the PR introduces a behavioral change if the execution of TS exhibits two different sets of states S and S', for respectively the *base* branch and the *Pull request* branch.

2.4.4 Behavioral Change Detection

We define behavioral change detection as the ability of a test suite to distinguish two versions of the same program. We approximate this detection by the fact that the test suite passes on a given version but fails on the other, i.e.the verified states of the two versions of the same program, *base* branch and *Pull request* branch are different.

In this paper, we aim at enhancing the ability of a test suite to detect behavioral changes. To do this, we use test suite amplification because it exploits the existing test suite. Starting from high quality test methods will lead to better result rather than starting from scratch. This is why, we modify each techniques to make them test suite amplification, i.e., a process that automatically processes the program and the existing test suite in order to generate a better version of the test suite.

Based on the graphical signature introduced previously, we aim for the following improvement:

$$TS\begin{pmatrix} BASE \\ PR \end{pmatrix} = \begin{pmatrix} \checkmark \\ \times \end{pmatrix}$$

2.4.5 Preliminary results

We obtain promising result on 3 real world pull requests from GitHub, detailled in the next subsections.

Case study: protostuff#167

The first pull request we consider as case study is in an open-source project called protostuff, which provides advanced serialization features for Java. It has 1485 commits, 14 contributors, 686 star gazers, and 70K lines of code.

The considered pull request is protostuff#167[2], it modifies the `toString()` method of class `io.protostuff.ByteString`. Figure 2.4 shows a screenshot of the pull request. In this pull request, the developers change the String representation of the `ByteStrings` as follows: instead of using a plain String representation of the array of bytes, the representation is now a String that includes a hashcode of the `ByteString` and its size. First, we make sure that this pull request contains a behavioral change, because it contains a new test case that fails on the *base* version. We compute the graphical signature of the pull request of the *base* test suite:

$$TS\begin{pmatrix} BASE \\ PR \end{pmatrix} = \begin{pmatrix} \checkmark \\ \checkmark \end{pmatrix}$$

```
    public String toString()
    {
-       return toStringUtf8();
+       return String.format("<ByteString@%s size=%d>",
+               Integer.toHexString(System.identityHashCode(this)), size());
    }
```

Figure 2.4: Application change in pull request protostuff#167 in class ByteString.

[2] https://github.com/protostuff/protostuff/pull/167

Listing 2.1: Original test method of protostuff to be amplified

```
1 void testCompareVsOther() {
      Bar testBar = new Bar(22,
3         ByteString.wrap("fuck yo test".getBytes()),
          false
5     );
      ... // assert later on
7 }
```

Listing 2.2: New test method of protostuff obtained with Assertion-amplification

```
void testCompareVsOther_amplified() {                                    1
    Bar testBar = new Bar(22,
        ByteString.wrap("fuck yo test".getBytes()),                      3
        false
    );                                                                   5
    assertEquals(22, testBar.getSomeInt());
(*@\xmark@*)  assertEquals("fuck yo test", testBar.getSomeBytes().toString());   7
        assertFalse(testBar.getSomeBoolean());
}
```

Now, we see whether a behavioral change detection technique is able to detect this modification. For this, we apply both Assertion-amplification and Input-amplification to the test `testCompareVsOther` of the test class `LowCopyProtobufOutputTest`. The first task to do is to select the test to be amplified: here the application change is in class `ByteString`, so we select the tests that use at least one method of class `ByteString`. The selected test is `testCompareVsOther`, of which an excerpt is shown in Listing 2.1. In this test, the method wrap from class ByteString is used. Note that the potentially offending string literals are those of the developers.

From this test, Assertion-amplification generates a new test case with 3 new assertions in 2.7 minutes: two `assertEquals` and one `assertFalse`, that are shown in Listing 2.2. The second `assertEquals` in line 7 is the one that can detect the behavioral change: it captures the behavior of the *base* branch in which the String representation of a `ByteString` is a direct translation of the `ByteString`'s content. This assertion passes on the *base* branch and fails on the new behavior proposed in *Pull request* branch.

This is a success for Assertion-amplification: this automatically generated test is able to capture the behavioral change. Input-amplification also captures this change, since it supersedes Assertion-amplification. As a result, the pull request signature is the expected one, the same test suite TS highlights the behavioral difference between the *base* version and the pull request:

$$ATS\begin{pmatrix} BASE \\ PR \end{pmatrix} = \begin{pmatrix} \checkmark \\ \times \end{pmatrix}$$

Case study: javapoet#550

The second chosen pull request is from a project named Javapoet. Javapoet is a library for generating Java source code. It has 755 commits, 62 contributors, has 7K lines of code and has been starred 5398 times. Pull request javapoet#550[3] adds not-null checks in the constructor of the `Builder` class and in method `addModifiers`. When the given parameter is `null`, the program throws an exception with a specific message. In the pull request, the developers add information in the exception flow

[3] https://github.com/square/javapoet/pull/550

Listing 2.3: Original test method of javapoet to be amplified

```
@Test
void nullExceptionsAddition() {
    try {
      MethodSpec.methodBuilder("doSomething").addExceptions(null);
      fail();
    } catch (IllegalArgumentException expected) {
      assertThat(expected).hasMessage("exceptions == null");
    }
}
```

in order to ease debugging. The graphical signature of this pull request for the *base* test suite is the following, which means that there is no test case that specifies the behavioral change:

$$TS\begin{pmatrix} BASE \\ PR \end{pmatrix} = \begin{pmatrix} \checkmark \\ \checkmark \end{pmatrix}$$

```
    private Builder(String name) {
+     checkNotNull(name, "name == null");
      checkArgument(name.equals(CONSTRUCTOR) || SourceVersion.isName(name),
          "not a valid name: %s", name);
      this.name = name;
@@ -320,6 +321,7 @@ public Builder addAnnotation(Class<?> annotation) {
    }

    public Builder addModifiers(Modifier... modifiers) {
+     checkNotNull(modifiers, "modifiers == null");
      Collections.addAll(this.modifiers, modifiers);
      return this;
    }
```

Figure 2.5: Proposed changes in pull request javapoet#550.

We now see if behavioral detection techniques work on this pull request. We apply Assertion-amplification and Input-amplification to the test method *nullExceptionsAddition* of `MethodSpecTest`, shown in Listing 2.3. This test specifies that when the parameter of the method `addExceptions` is `null`, the program throws an exception with a specific message (call to method `hasMessage`).

First, we apply Assertion-amplification and notice that Assertion-amplification does not succeed to detect the behavioral change. Next, we apply Input-amplification. For this pull request, Input-amplification manages to produce a test that specifies the behavioral change in 20 minutes. It results in the amplified test method shown in Listing 2.4.

The amplification applied to obtain this valuable new test method is as follow: 1) Input-amplification replaces the literal string `"doSomething"` by a `null` value; 2) Assertion-amplification executes the transformed test method 3) since the method now throws an exception with this new parameter, Assertion-amplification wraps the body with a try/catch block (to catch a `NullPointerException`). It adds a fail statement at the end of the body of the try, i.e., it expects that the exception is thrown, and adds an assertion on the message of the exception. The message is null in this case, since the exception is generated by the JVM.

When executing this automatically generated test method on the *Pull request* branch, it fails, while it passes on the *base* branch, which proves that the behavioral change has been specified. The `assertEquals(null, expected_1.getMessage())` in Line 10, marked by ✗, captures the behavioral changes since the message of the exception is no more null. Finally, the amplified test suite (ATS) has the correct test signature:

Listing 2.4: New test method of javapoet(#550) obtained with Input-amplification

```
1  @Test
   void nullExceptionsAdditionlitString2_failAssert0() {
3      try {
           try {
5              MethodSpec.methodBuilder(null).addExceptions(null);
           } catch (IllegalArgumentException expected) {
7          }
           fail("nullExceptionsAdditionlitString2 should have thrown
               NullPointerException");
9      } catch (NullPointerException expected_1) {
   (*@\xmark@*)         assertEquals(null, expected_1.getMessage());
11     }
   }
```

$$ATS\begin{pmatrix} BASE \\ PR \end{pmatrix} = \begin{pmatrix} \checkmark \\ \times \end{pmatrix}$$

Case study: javapoet#608

The third and final case study of this paper is about pull request javapoet#608[4], from the same project javapoet as the previous case study. As shown in Figure 2.6, this pull request modifies the method `zeroWidthSpace` from the `LineWrapper` object. This method is responsible to insert a new line and to update the indentation level of a Java source file. The pull request aims at fixing the behavior as follows: if the current line has no indentation, i.e., the `column` is equal to zero, the method should directly return. This pull request does not specify the behavioral change and consequently, the test signature is:

$$TS\begin{pmatrix} BASE \\ PR \end{pmatrix} = \begin{pmatrix} \checkmark \\ \checkmark \end{pmatrix}$$

```
    void zeroWidthSpace(int indentLevel) throws IOException {
      if (closed) throw new IllegalStateException("closed");

+     if (column == 0) return;
      if (this.nextFlush != null) flush(nextFlush);
      this.nextFlush = FlushType.EMPTY;
      this.indentLevel = indentLevel;
```

Figure 2.6: Proposed changes in pull request javapoet#608.

We apply Assertion-amplification and Input-amplification on test method `wrapEmbeddedNewlines_ZeroWidth` from `LineWrapperTest`, because they both invoke method `zeroWidthSpace`, shown in Listing 2.5

Only Input-amplification succeeds to generate an amplified test method that detect the behavioral changes. This amplified test method is shown in Listing 2.6.

In 2.11 minutes Input-amplification is successful in detecting the behavioral change, by applying the following transformations. First, it removes both method calls on Lines 5 and 7. Then, it adds assertions on observable values of the variable `out`. It results in a single assertion, on line 8 marked

[4] https://github.com/square/javapoet/pull/608

Listing 2.5: Original test method of javapoet to be amplified for #608

```
@Test
public void wrapEmbeddedNewlines_ZeroWidth() {
    StringBuffer out = new StringBuffer();
    LineWrapper lineWrapper = new LineWrapper(out, "  ", 10);
    lineWrapper.append("abcde");
    lineWrapper.zeroWidthSpace(2);
    lineWrapper.append("fghijk\nlmn");
    lineWrapper.append("opqrstuvwxy");
    lineWrapper.close();
    assertThat(out.toString()).isEqualTo("abcde\n    fghijk\nlmnopqrstuvwxy");
}
```

Listing 2.6: New test method of javapoet(#550) obtained with Input-amplification

```
@Test
public void wrapEmbeddedNewlines_ZeroWidth_rm147_rm598() {
    StringBuffer out = new StringBuffer();
    LineWrapper lineWrapper = new LineWrapper(out, "  ", 10);
    lineWrapper.zeroWidthSpace(2);
    lineWrapper.append("opqrstuvwxy");
    lineWrapper.close();
(*@\xmark@*)  assertEquals("\n    opqrstuvwxy", out.toString());
}
```

by ✗, which captures the behavioral difference between the two versions of the program (he *base* and the *Pull request* branches). After the *Pull request* changes, the expected value of `out.toString()` is "opqrstuvwxy" without new line character and indentation. As expected, the test signature of the amplified test suite (ATS) shows that the behavioral change is now specified:

$$ATS\begin{pmatrix}BASE\\PR\end{pmatrix} = \begin{pmatrix}\checkmark\\ \times\end{pmatrix}$$

2.5 Future work

As ultimate goal, we aim at integrating DSpot with Continuous Integration services such as Jenkins and Travis: at each change pushed by the developers, the CI would trigger DSpot in a specific configuration, amplify the test suite and report the results to the developers. We can imagine several scenario such as: the detection of regression detection bugs, or online enhancement of the test suite. The former aims at temporarily increasing the ability of the test suite to detect regressions introduced by changes. Such a process is almost mandatory in the context of pull request development scenario, to ensure that, before merging the changes, they do not introduce any undesired behavior. Thus, the amplification would help reviewers of pull request to ensure this. The latter aims at improving the test suite permanently, such as we did in the Pull Request Evaluation. The idea is that developers analyze the amplified test, and add them to the test suite.

In any case, integrating DSpot in the continuous integration services requires to select the tests to be amplified to scale and speed-up the amplification. To do this, we can rely on the huge literature [3] on test minimization, selection and prioritization.

Chapter 3

Descartes

Mutation testing [1] is a technique that assesses the quality of a test suite by introducing artificial faults in the code and then verifying of the test cases can detect those faults. Extreme mutation, proposed by Niedermayr and colleagues [2], is an alternative to traditional mutation testing that performs coarse-grained transformations. This approach eliminates, at once, all the effects of a method. For a `void` method, it removes all the instructions in the body. If the method is not `void`, then the body is replaced by a single return instruction with a predefined and constant value. In this way the method is guaranteed to produce always the same result.

In addition to a mutation score, as the traditional approach, extreme mutation provides a list of worst tested methods. With the help of this technique is possible to detect methods that has been executed by the test suite but where no extreme mutant is detected by any test case. These methods are qualified as **pseudo-tested** in the work of Niedermayr et al.

In this chapter we describe the development progress of Descartes, a tool developed in the context of the STAMP project that uses extreme mutation to automatically detect pseudo-tested methods in Java programs. We also summarize some of the main experiments we have performed with the help of Descartes to explore the nature of pseudo-tested methods.

3.1 Development progress

Descartes has been conceived as a mutation engine for PITest, which is a mature, open-source mutation testing tool that targets Java programs and provides support for all major build systems. A mutation engine is a plugin for PITest that handles the discovery and creation of mutants. The remaining parts of the PITest framework take care of selecting and running the tests against the mutants proposed by the mutation engine.

Descartes classifies a method in pseudo-tested if no extreme transformation is detected, partially-tested if the method has both detected and undetected transformations and required, if all transformations were detected. These two last categories are a refinement of the “tested” category in the work of Niedermayr et al.

From the pre-release of version 0.2 in November 2017 to the moment in which this document is being written, the Github repository of Descartes shows 119 additional commits to the master branch. These changes have been mostly directed to: 1) optimize the code; 2) tackle several bugs and 25 issues reported in the tracker (47 since the beginning of the project); 3) keep the code up-to-date with the regular releases and API changes of PITest; and 4) the incorporation of new features to make it more useful and easier to adopt by developers.

At the moment, Descartes provides several custom reporting extensions. In addition to the previous general JSON report format, now the engine is shipped with two new custom reporters that produce the list of pseudo-tested methods found in the code base. One of such reporters also produces a JSON

Table 3.1: List of project, number of methods under analysis and number of pseudo-tested methods

Project	#MUA	#PSEUDO	PS_RATE	Project	#MUA	#PSEUDO	PS_RATE
authzforce	291	13	4%	jgit	2,539	296	12%
aws-sdk-java	1,800	224	12%	joda-time	2,526	82	3%
commons-cli	141	2	1%	jopt-simple	256	2	1%
commons-codec	426	12	3%	jsoup	751	28	4%
commons-collections	1,232	40	3%	sat4j-core	585	143	24%
commons-io	641	29	5%	pdfbox	2,241	473	21%
commons-lang	1,889	47	2%	scifio	158	72	46%
flink-core	1,814	100	6%	spoon	2,938	213	7%
gson	477	10	2%	urbanairship	1,989	28	1%
jaxen	569	11	2%	xwiki-rendering	2,049	239	12%
jfreechart	3,496	476	14%	Total	28,808	2,540	9%

file with the list of the methods that have been mutated and for each method it contains the mutations that have been performed and the final categorization in pseudo-tested, partially-tested or required. This report is meant to support automation tasks. The other reporter produces a set of HTML files containing human-readable information about the pseudo-tested and partially-tested methods. This report should help developers to understand the testing issues behind the signaled methods.

As for version 0.2 Descartes had the ability to detect and skip from the analysis a set of methods that are generally not in the interest of developers and therefore could be seen as false positives. This functionality has been rewritten in newer versions to increase the categories of methods to skip and make this feature configurable. Since version 1.2, Descartes can be configured to skip empty `void` methods, methods generated by the compiler, `toString` and `hashCode` methods, simple getters and setters, methods that simply return literal constants and static class initializers.

To this date, Descartes has been released six more times and all these versions are available from Maven Central in the form of compiled artifacts[1].

3.2 How frequent are pseudo-tested methods?

Pseudo-tested methods are intriguing. They are covered by the test suite, their body can be drastically altered and yet no test case notices the transformations. Niedermayr et al [2] find such methods in all their study subjects even in those with high coverage ratios.

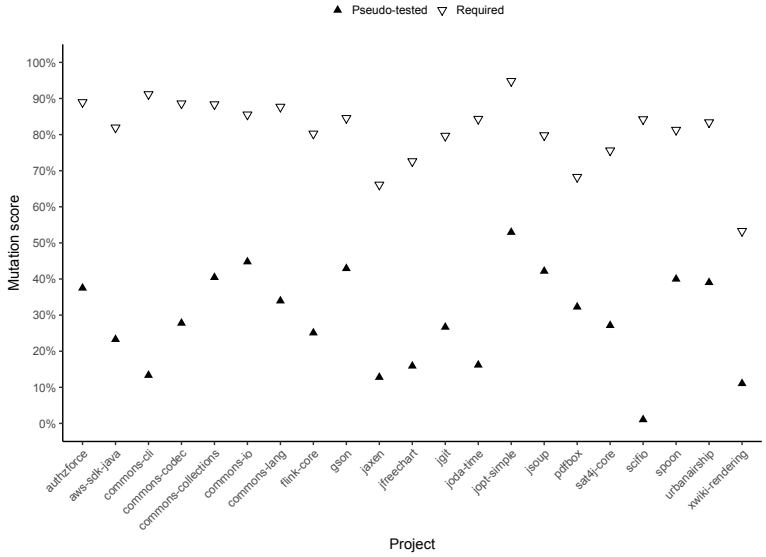
To further investigate these methods we perform an empirical study based on the analysis of 21 open-source projects. These are the same projects used in previous experiments and can be checked in tables 3.1 and 3.2 of D1.2.

Table 3.1 shows the list of projects included in our experiments. The *#MUA* column shows the number of methods targeted in our analysis. That is, methods covered by the test suite, that are not empty, not generated by the compiler, not marked as deprecated, they are not simple getters or setters, constructors or static initializers. Methods that only return a literal constant are excluded as well. The *#PSEUDO* column shows the number of pseudo-tested methods we have found and *PS_RATE* shows the ratio of *#PSEUDO* to *#MUA*.

This first analysis is a conceptual replication of the work of Niedermayr et al that confirms one of their main findings. We find pseudo-tested methods in all our study subjects. This replication miti-

[1] https://mvnrepository.com/artifact/eu.stamp-project/descartes

Figure 3.1: Mutation score for mutants placed inside pseudo-tested and required methods.



gates external threats to the validity of their study by using our own tools and a different set of projects. The number of pseudo-tested methods in our analysis ranges from two to 476. As for the PS_RATE, 14 projects show a percentage below 10%, four are below 20 % only three have percentages above 20% and only one of them reaches a percentage above 40%. In general, the ratio of pseudo-tested methods is low.

3.3 Are pseudo-tested methods the weakest points in the program with respect to the test suite?

Test suites fail to asses the presence of any effect in pseudo-tested methods. As such, these methods can be considered as badly tested, even though they are covered by the test suite. To further confirm this fact we assess the test quality of these methods with the traditional mutation score as a test adequacy criterion.

For each of our study subjects we run a mutation analysis based on *Gregor*, PITest's default mutation engine. In this new analysis we used the standard mutation operators supported by PITest. We extract the set of mutants that have been placed in the body of pseudo-tested methods and the set of mutants placed in the body of required methods. For both sets we compute a mutation score, that is, the ratio of detected mutants, namely MS_pseudo and MS_required respectively.

Figure 3.1 shows the results of this experiment. In all cases, the mutation score of pseudo-tested methods is significantly lower than the score of normal required methods. This means that a mutant planted inside a pseudo-tested method has more chances to survive than a mutant planted in required methods. To validate this graphical finding, we compare MS_pseudo and MS_req with the Wilcoxon statistical test and we find significant evidence of a difference with a p-value $p < 0.01$ and a large effect size of 1.5. It calls the attention, that, in no case, MS_pseudo score was 0%. So, even when extreme transformations are not spotted by the test suite, some traditional mutants inside these methods can be detected.

3.3.1 Are pseudo-tested methods relevant for developers to improve the quality of the test suite?

To check if pseudo-tested methods could be relevant hints to improve a test suite, we manually analyzed `void` and `boolean` pseudo-tested methods, which are accessible from an existing test class and have more than one statement. We identify eight testing issues revealed by these pseudo-tested methods: two cases of a miss-placed oracle, two cases of missing oracle, three cases of a weak oracle and one case of a missing test input. These issues have been found in seven of our study subjects.

For each testing issue we prepare a pull request with a fix, or we send the information by email to the development team. We collect qualitative feedback from the development teams about the relevance of the testing issues revealed by pseudo-tested methods.

We now briefly summarize the discussion about each testing issue.

`aws-sdk-java`: We made a pull request (PR)[2] to explicitly assess the effects of one pseudo-tested method covered by four test cases. The oracle in the test cases was miss-placed inside a mock object. When the method is emptied, then the assertion is never verified. The pull request was accepted.

`commons-collections`: We made a PR[3] to target five implementations of iterator operations. In the corresponding classes, these operation should not be supported and therefore they throw an exception. The test cases covering these methods were built in a way that, if the methods are emptied, then the test cases do not fail. This is another case of a miss-placed assertion. The pull request was accepted.

`commons-codec`: We made a PR[4] to assess the effects of a method covered by only one test case. The test case in question had no assertion. When the assertion was added one of the inputs turned to be incorrect. We proposed a fix for this as well. The PR was merged and increased the code coverage by 0.2%.

`commons-io`: We made a PR[5] to assess the effects of several methods in a stream implementation. This stream should broadcast its content to two other streams. The test cases verified that the output was the same, but not that both outputs were correct. Our fix tackled this and made the assertion stronger. The PR was accepted and slightly increased the code coverage by 0.07%.

`spoon`: We sent an email to the developers regarding one method covered by only one test case. The method seemed to have an important role in the class is was declared. The developers expressed that it should be better specified by adding stronger verifications. They took no immediate action but opened a general issue [6].

`flink-core`: We sent an email regarding one method covered by only one test case. The assertion in the test case actually verifies that the method has no wrong side effect. In our opinion, the oracles should also verify that the main effect of the method is achieved. In this case, if the method is emptied the test case does not fail. The developers took no action to target the issue.

`sat4j-core`: We sent an email regarding two `void` methods. One of them was covered by only one test case to target a specific bug and avoid regression issues. The other method was covered by 68 different test cases. The lead developer considered the first method as helpful to realize that more assertions were needed in the covering test case. Consequently, he made one commit[7] to verify the behavior of this method. The second pseudo-tested method was considered a bigger problem, because it implements certain key optimizations for better performance. No test case was difficult enough to witness the effect of such optimization in the test suite. Consequently, the developer made a new commit[8] with an additional, more difficult, test case to target the issue.

[2] https://github.com/aws/aws-sdk-java/pull/1437
[3] https://github.com/apache/commons-collections/pull/36
[4] https://github.com/apache/commons-codec/pull/13
[5] https://github.com/apache/commons-io/pull/61
[6] https://github.com/INRIA/spoon/issues/1818
[7] https://gitlab.ow2.org/sat4j/sat4j/commit/46291e4d15a654477bd17b0ce905926d24e042ca
[8] https://gitlab.ow2.org/sat4j/sat4j/commit/afab137a4c1a54219f3990713b4647ff84b8bfea

Table 3.2: The pseudo-tested methods systematically analyzed by the lead developers, through a video call.

Project	Sample size	Worth	Percentage	Time spent (HH:MM)
`authzforce`[11]	13 (100%)	6	46%	29 min
`sat4j-core`[12]	35 (25%)	8	23%	1 hr 38 min
`spoon`[13]	53 (25%)	16	23%	1 hr 14 min
Total	101	30	30%	3 hr 21 min

From this exercise we can conclude that 1) all developers agreed that it is easy to understand the problems identified by pseudo-tested methods. This is confirmed by the fact that, we, as outsiders to those projects, with no knowledge or experience, can also grasp the issue and propose a solution. The developers acknowledged the relevance of the uncovered flaws; 2) when developers were given the solution for free (through pull requests written by us), they accepted the test improvement; 3) when the developers were only given the problem, they did not always act by improving the test suite. They considered that pseudo-tested methods provide relevant information, and that it would make sense to enhance their test suites to tackle the issues. But they do not consider these improvements as a priority. More details about this interaction with development teams can be found online[9].

Apart from the aforementioned issues, the XWiki development team has been able to incorporate four changes (commits) with improvements spotted with the use of Descartes. These changes fix an error in one equality comparison, a missing assertion, missing input for a method that always returned $false$ in the test cases and notably the removal and simplification of part of the code base. The changes were added to `xwiki-commons` and `xwiki-rendering`. More details can be found online [10].

3.4 Which pseudo-tested methods do developers consider worth an additional testing action?

We set up three independent video calls with the teams of `authzforce`, `sat4j-core` and `spoon`. The goal was to discuss with them a set of pseudo-tested and grasp the criteria used by developers to consider a method as relevant enough to trigger additional work on the test suite. Before each call we prepared, and made available, a report with a list of pseudo-tested methods, the extreme transformations that were used and the test cases that covering those methods. For `sat4j-core` and `spoon`, we randomly choose 25% of all pseudo-tested methods. For `authzforce` we presented all 13 pseudo-tested methods.

For each method, we asked the developers to determine if: 1) given the structure of the method, and, 2) given its role in the code base, they consider it is worth spending time creating new test cases or fixing existing ones to specify those methods. We also asked them to explain the reasons behind their decision.

Table 3.2 shows the number of pseudo-tested methods coming from each sample, the number of methods developers considered relevant along with footnotes with links to the online summary of the interviews. we also show the time spent in the discussion.

The percentage of relevant methods is 23% for `sat4j-core` and `spoon`, For `authzforce` the percentage of methods to be specified is 46%, but the absolute number (6) is rather low. This

[9] https://github.com/STAMP-project/descartes-experiments/blob/direct-communications.md
[10] https://github.com/STAMP-project/descartes-usecases-output/tree/master/xwiki
[11] https://github.com/STAMP-project/descartes-experiments/blob/master/actionable-hints/authzforce-core/sample.md
[12] https://github.com/STAMP-project/descartes-experiments/blob/master/actionable-hints/sat4j-core/sample.md
[13] https://github.com/STAMP-project/descartes-experiments/blob/master/actionable-hints/spoon/sample.md

indicates that, potentially, many pseudo-tested come from functionalities considered less important or not a priority, therefore not well tested.

Worthless specifying: Developers could consider a pseudo-tested as useless to test or *not important*, if it meets one of the following criteria:

- The code has been automatically generated.
- The method is part of debugging functionalities like formatting a debug or log message or creating and returning an exception message or an exception object.
- The method is part of features that are not widely used in the code base or in external client code.
- The method has not been deprecated but its functionality is being migrated to another interface.
- The code of the method is considered as simple or trivial to need a specific test case.
- Methods used as placeholders for unneeded members in interface implementations.
- Receiving methods in a delegation pattern that add little or no logic when invoking the delegate.

Worth specifying with additional tests: On the other hand, developers provided the following reasons when considering a pseudo-tested important enough:

- A method that supports a core functionality of the project or part of the main responsibility of the declaring class.
- A method supporting a functionality that is widely used in the code base. It could be the method itself that is being frequently used or the class that declares the method.
- A method known to be relevant for external client code.
- A new feature that is partially supported or not completed yet.
- Methods which are the only possible way to access certain features of a class.
- Method verifying preconditions as they guarantee the integrity of the operations to be performed.

We have observed cases where a method meets criteria to be worth of specification and at the same time to be worthless of additional testing actions. The final decision of developers in those cases is subjective and responds to their internal knowledge about the code base. This means that it is difficult to devise an automatic procedure able to automatically determine which methods are worth of additional testing actions.

3.5 Mutation Analysis Survey

The work on test assessment with Descartes, is complemented by the survey on mutation testing, recently published in the Journal of Software Testing and Verification [5]. In this work, we provide a systematic literature review on the application perspective of mutation testing based on a collection of 191 papers published between 1981 and 2015. In particular, we analyzed in which quality assurance processes mutation testing is used, which mutation tools and which mutation operators are employed. Additionally, we also investigated how the inherent core problems of mutation testing, ie, the equivalent mutant problem and the high computational cost, are addressed during the actual usage. The results show that most studies use mutation testing as an assessment tool targeting unit tests, and many of the supporting techniques for making mutation testing applicable in practice are still underdeveloped. Based on our observations, we made 9 recommendations for future work, including an important suggestion on how to report mutation testing in testing experiments in an appropriate manner.

3.6 Future work

With our experiments, we have confirmed the wide presence of pseudo-tested methods among projects in different application domains, development practices and size. We have also assessed the relevance of pseudo-tested methods as concrete hints to reveal weak test oracles, fact confirmed by the developers who accepted the pull requests we proposed. On the flip side of the coin, we have observed that less than 30% of pseudo-tested in a sample from three projects could be considered as priorities by developers. These results are currently under review for the Empirical Software Engineering Journal [4].

In the light of these facts the immediate step is to investigate automatic test generation techniques targeted towards these methods. This line should also consider ways to prioritize pseudo-tested methods, given the criteria collected from developers. Other lines of work should be directed to improve the usability of Descartes guided by the requests and issues reported by the other project partners. Some of these issues are more related to the current limitations and capabilities of PITest, for example, supporting custom JUnit test runners. Addressing this kind of issue could be also a contribution to the broader community of PITest's users.

Chapter 4

Flaky and Clover

4.1 Flaky Test Tool

A tool was developed to handle Flaky Tests and prevent false positive emails. It was specifically developed for Jenkins (as a pipeline) and JIRA but can easily be adapted to other issue trackers.

Here is how it works:

- When your CI runs the jobs to build your code and when a test fails, the Flaky Test Tool will check if the failing tests are flickers or not.
- If all the tests are flickers then no email is sent to the developers thus avoid false positives. This is done by the Flaky Test Tool by querying the JIRA instance and gathering the list of flickering tests, using a custom field (see below).
- the Flaky Test Tool also handles false positives related to environmental issues such as a Git server not responding, an X display not being present, etc. This is done by looking for specific regexes in the build result logs.
- In addition the Flaky Test Tool modifies the Jenkins Job page to report the Test as being a flicker and the job as containing flickering tests. This is illustrated in Figure 4.1.
- It remains the responsibility of the developers to identify flaky tests and register them in JIRA, using the format: `<FQN name of test class>#<test name>`. For example: `org.xwiki.platform.notifications.test.ui.NotificationsIT#testCompositeNotifications`

Here's how it can be used:

- You need to add a custom field in your JIRA instance. Here iŹs for example the one added in the XWiki JIRA instance. Cf. Figure 4.2
- Then you need to copy paste the script and integrate it into your Jenkins pipeline: https://github.com/STAMP-project/stamp-flaky-tool/blob/master/build.groovy

4.2 Enhanced Clover Tool

XWiki has developed [1] some script to help automate the gathering of coverage reports and to help compare them together, and to even fail the build if total coverage goes down.

[1] http://massol.myxwiki.org/xwiki/bin/view/Blog/FullCoverageClover

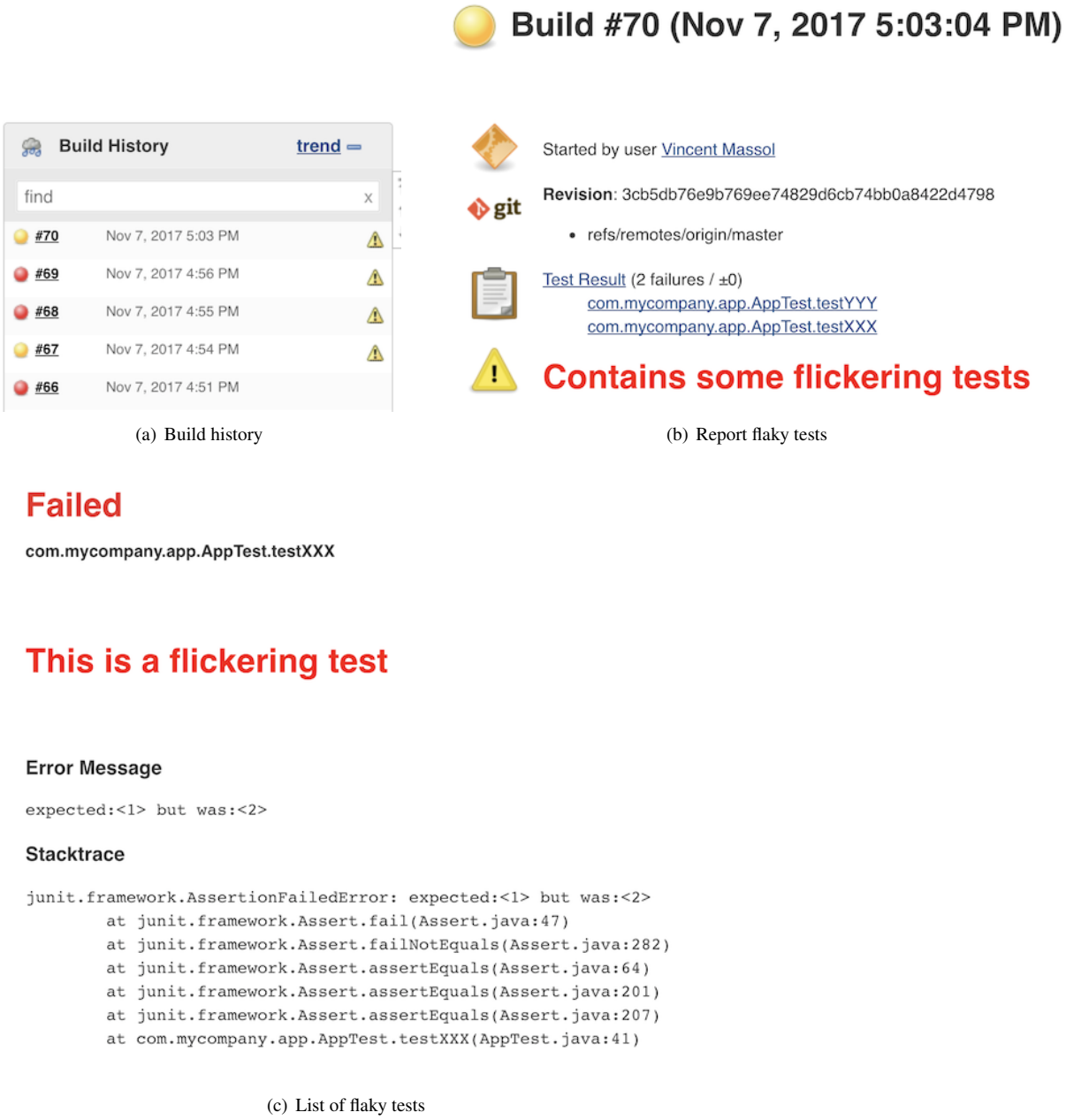
Specifically the idea is to have the CI run some job regularly (on the XWiki project we run them once per month) that will compute the full test coverage using Open Clover, taking into account coverage generated by UI tests and more generally by modules indirectly exercising code from other modules. We also support covering code that is spread on various source repositories (e.g. various Git or GitHub repositories).

The implementation [2] is done as a Jenkins pipeline and has to be adapted for each project.

At a high level the tool does the following:

- For each repo of the project, run the project's build with Clover. Note that Clover must be enabled in the project's build as a prerequisite.
- Then results are analyzed: the tool looks for the last passing report (if none are found then consider the new report as the baseline). Each module's Test Percentage Coverage (TPC) is computed and compared with the previous value from the previous reports and if one or several modules have lowered TPCs then the build fails and an email is sent to developers.
- Note that tests are excluded from TPC computations in order to harmonize the results. It is also not generally interesting to measure coverage for tests themselves.
- In addition a colored report is generated showing the evolution of each module's TPC.
- Results are all published on a web site for later perusal. For example
  - reports for the XWiki project: http://maven.xwiki.org/site/clover/
  - diff report: https://github.com/STAMP-project/stamp-clover-tool/blob/master/samples/xwiki/Clover-2017-11-09-2018-04-03.pdf

[2] https://github.com/STAMP-project/stamp-clover-tool



(a) Build history

(b) Report flaky tests

(c) List of flaky tests

Figure 4.1: Jenkins report of the flaky tool

Figure 4.2: XWiki Jira configuration to use the flaky tool

Chapter 5

Industrialization of Unit Test Amplification

The WP1 tools (DSpot and Descartes) implement the core features of test amplification and mutation testing. Yet, to be usable from an industrial point of view, and especially by the use case providers, those tools need to be integrated in several build chains and CI environments. For this, we develop additional tools, related to DSpot and Descartes. Those tools are:

- DSpot Maven plugin: a Maven plugin to run test amplification thanks to the goal dspot:amplify-unit-tests. Source code available at https://github.com/STAMP-project/dspot/tree/master/dspot-maven
- Jenkins DSpot plugin: a Jenkins plugin which makes it possible to run DSpot as a build step within a Jenkins pipeline or freestyle job. Source code available at https://github.com/STAMP-project/dspot-jenkins-plugin
- Jenkins Descartes plugin: a Jenkins plugin which makes it available trend reports about mutation coverage obtained applying iteratively Descartes on a Java project. Source code available at https://github.com/STAMP-project/jenkins-stamp-report-plugin
- STAMP IDE: two Eclipse plugins have been developed to offer wizards to configure DSpot and Descartes execution over Java projects. Configurations can be saved and used later. Execution logs are made available in the Eclipse console. Source code available at https://github.com/STAMP-project/stamp-ide
- DSpot support for Gradle Java projects: an AutomaticBuilder interface has been implemented in order to support Java projects which use Gradle as build system. Source code available within DSpot code base at https://github.com/STAMP-project/dspot/tree/master/dspot/src/main/java/eu/stamp_project/automaticbuilder
- PitMP: a maven plugin to run Pitest and Descartes on multi-module projects. Source code available at https://github.com/STAMP-project/pitmp-maven-plugin
- STAMP microservices: REST API have been designed to be implemented as microservices, in order to provide developers with STAMP as a Service

5.1 The DSpot Maven plugin

Introducing a new tool in the developer daily life only succeeds if its usage is quite familiar and similar to other tools the developer uses. Furthermore if the new tool introduces a really new concept as test

amplification, it is crucial to have it in a simple manner. At the same time, if the new tool is meant to be used also in a CI system, a proper wrapper is needed for this purpose. Apache Maven is a tool and a technology that allows for having both scenarios (individual productivity and CI pipelines), and this led to the need for a Mavel plugin exposing a specific "test amplifcation" goal.

5.1.1 How does it work?

The DSpot Maven plugin exposes a specific goal for test amplification. In this way a developer can run DSpot simply executing Maven as usual, providing DSpot input parameters as Maven parameters, or simply storing them in a properties file, which then is provided by the means of a Maven parameter to the plugin. In the same way, the Maven plugin can be used to configure a Jenkins pipeline in which a specific test amplification step is added to the ordinary CI pipeline.

5.1.2 Development status

The DSpot Maven plugin is aligned to the current DSpot version and acts as a simple wrapper of DSpot.

5.1.3 Future work

- Keep the code aligned with DSpot evolution
- Optimize the usage of parameters (at the moment several paramaters are passed to DSpot directly from a properties file).
- Add JUnit tests: at the moment, tests are done manually, just to check that DSpot invocation and test amplification starts and works as calling DSpot directly.

5.2 The Descartes Jenkins plugin

One of the most useful features a CI system offers is reporting. Usually reports are made available for a single execution, or rather showing trends for specific metrics. With Descartes it's possible to assess the mutation detection capabilities of test suites and provide developers with useful information to increase their robustness. Having this feature embedded in a CI system makes more powerful and effective its usage.

5.2.1 How does it work?

The Descartes Jenkins plugin provides developers with means to configure a Jenkins free-style job with a specific extreme mutation testing step, in order to assess the robustness of their test suites. The execution result is shown with several reports that show a detailed report for one single execution (in form of a pie chart with percentages of not-covered, tested, partially-trested and pseudo-tested methods, and the possibility to drill-down to every single method analysed by Descartes).

Another report is made available to show the trend of mutation coverage among several builds.

Access to a single execution report is also possible from a context menu single build:

5.2.2 Development status

The Descartes Jenkins plugin is quite stable and offers all the relevant information to fix and strenghten test suites in order to icnrease their mutation detection capabilities.

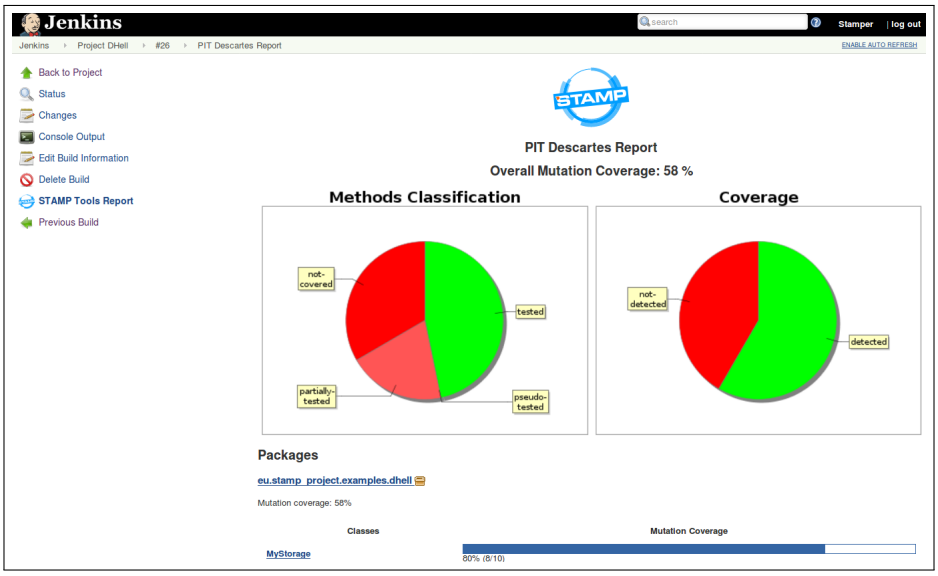


Figure 5.1: Screenshot of a Descartes execution report within a single Jenkins build

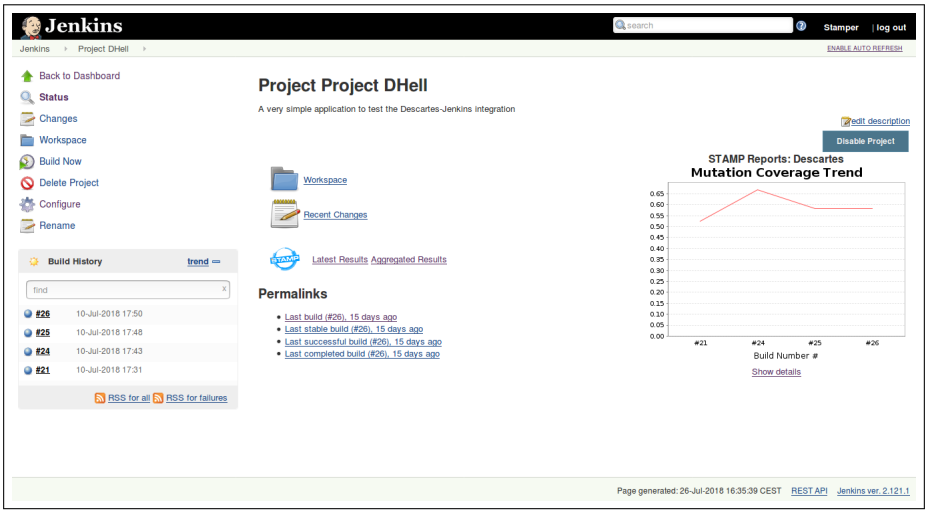


Figure 5.2: Screenshot of a Descartes trend report among several Jenkins builds

5.2.3 Future work

- extend the plugin to support also Jenkins pipelines
- Keep the code aligned with Descartes evolution
- Add test cases: at the moment, tests are done manually, just to check that Descartes execution within Jenkins works as expected, and reports are built accordingly to the execution results.

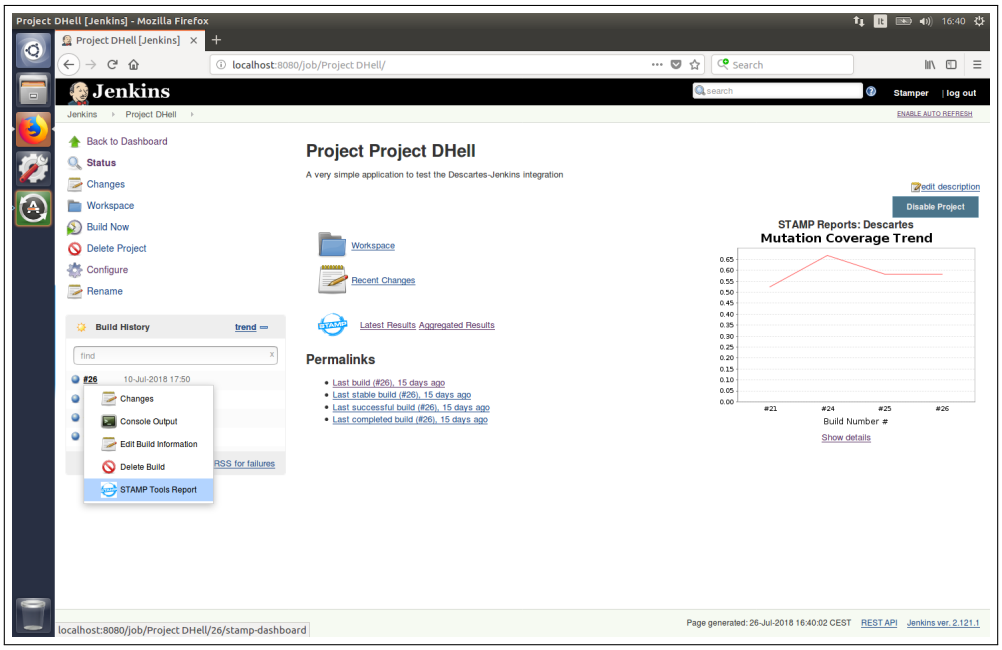


Figure 5.3: Screenshot of the context menu to access a Descartes execution from a Jenkins build

5.3 The STAMP IDE

One of the primary concerns of the STAMP consortium was to introduce STAMP novel features in the most typical development toolboxes and the natural choice was to focus on Eclipse IDE integration. Eclipse IDE offers a typical plugin-based architecture and this ease the process to integrate new functionalities making them available within the well-known Eclipse user experience. This led the consortium to develop two plugins that let developers to use DSpot and Descartes within their preferred development environment, making available graphical wizard to configure the tools, execute them and inspect the results.

5.3.1 How does it work?

The STAMP IDE offers a main and several contextual menu access for STAMP tools. Th access is context-sensistive: STAMP menus are activated only for Maven/Gradle Java projects.

A wizard is available to assist on DSpot and Descartes configuration, and multiple configurations can be stored and managed as well.

The execution log are made available within the Eclipse console.

5.3.2 Development status

DSpot and Descartes are both supported by STAMP IDE. STAMP IDE supports Eclipse Oxygen and newer versions (tested on Eclipse Photon).

5.3.3 Future work

- Keep the code aligned with DSpot and Descartes evolution
- Integration with issue trackers (at the moment Jira is the first choice).

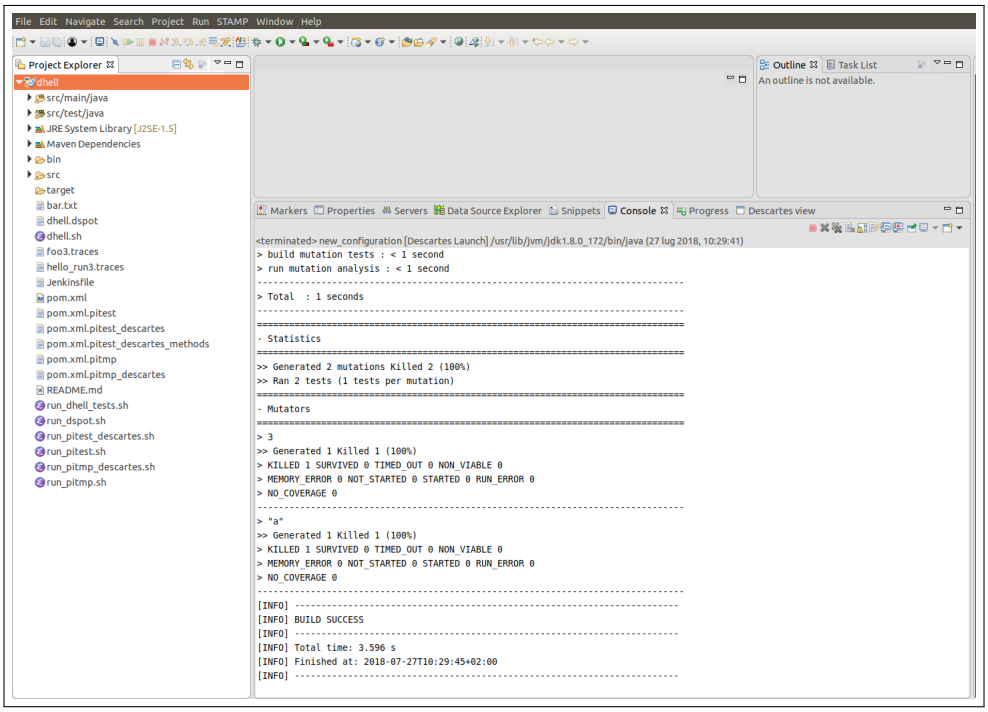


Figure 5.4: Screenshot of STAMP IDE showing an execution of Descartes and related results within the console

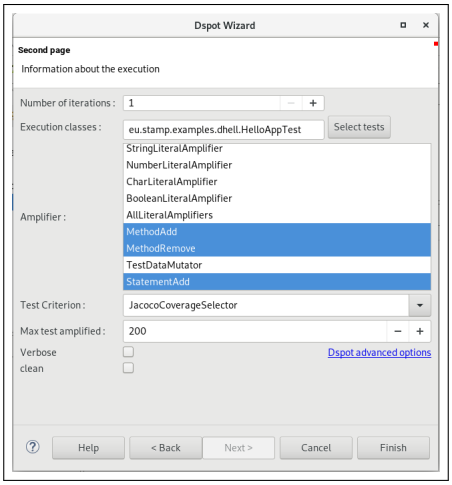


Figure 5.5: Screenshot of STAMP IDE showing an execution of DSpot configuration wizard

- STAMP dashboard.
- STAMP remote execution (possibility to execute DSpot and DEscartes remotely once DSpot and Descartes microservices will be available)

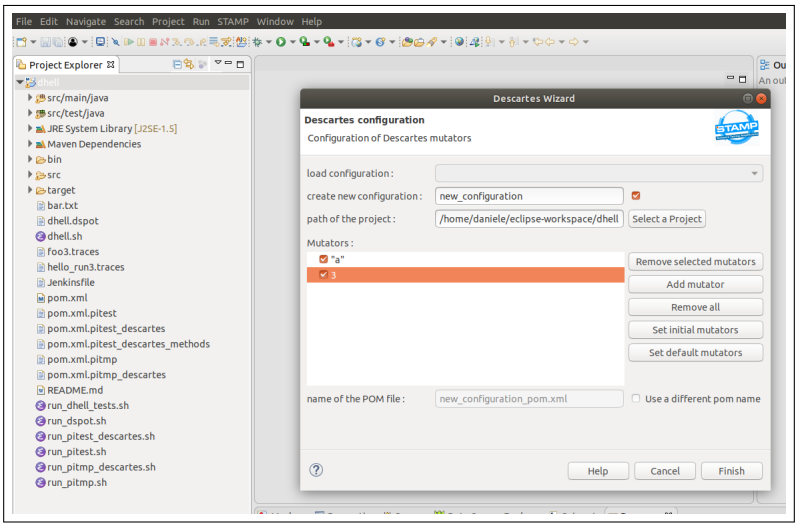


Figure 5.6: Screenshot of STAMP IDE showing an execution of Descartes configuration wizard

5.4 Gradle Java projects support for DSpot

Earliest version of DSpot supported just Java Maven-based projects. In order to introduce DSpot usage also within java projects which use Gradle ase build system, DSpot was refactored to support different build systems as well, making it available the eu.stamp_project.automaticbuilder.AutomaticBuilder interface. Potentially this interface allows for supporting several Java build systems. The native Maven support was refactored as an implementation of this interface (eu.stamp_project.automaticbuilder.MavenAutoma and then a new implementation supporting Gradle Java projects has been made available (eu.stamp_project.automaticbui

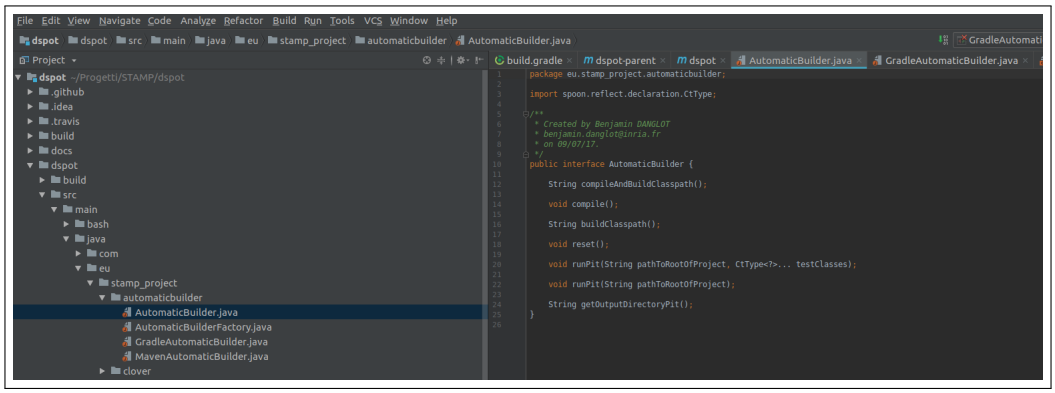


Figure 5.7: AutomaticBuilder interface

5.4.1 How does it work?

The GradleAutomaticBuilder implementation is based on Gradle Tooling official API which allow to execute programmatically several Gradle tasks.

All the methods exposed by the AutomaticBuilder interface has been implemented accordinglyThe

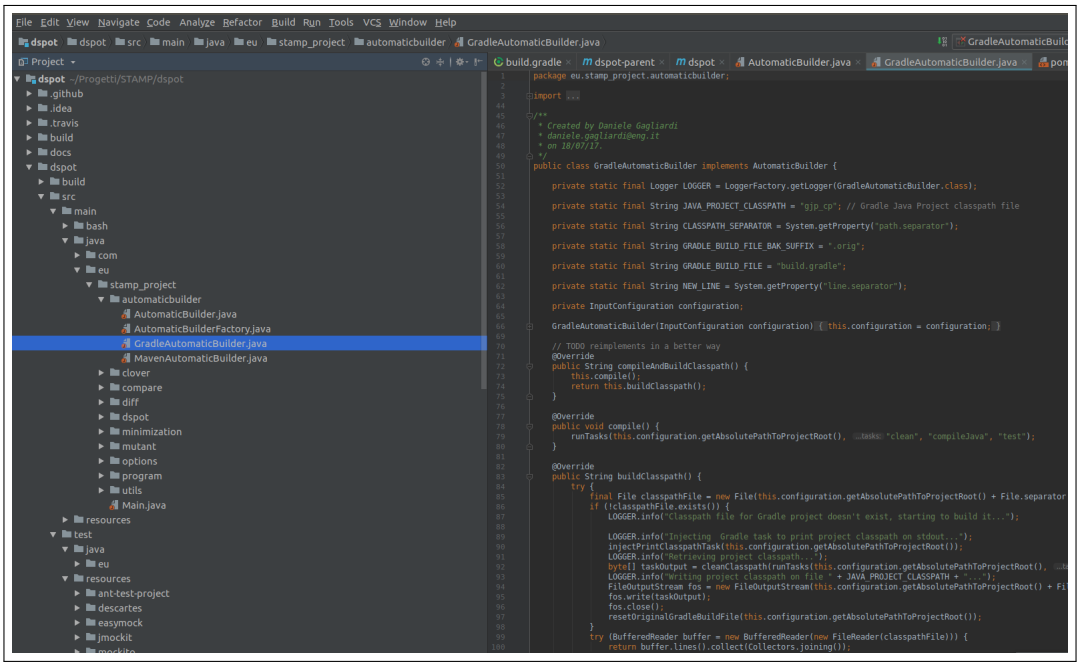


Figure 5.8: GradleAutomaticBuilder implementation

STAMP IDE offers a main and several contextual menu access for STAMP tools. Th access is context-sensistive: STAMP menus are activated only for Maven/Gradle Java projects.

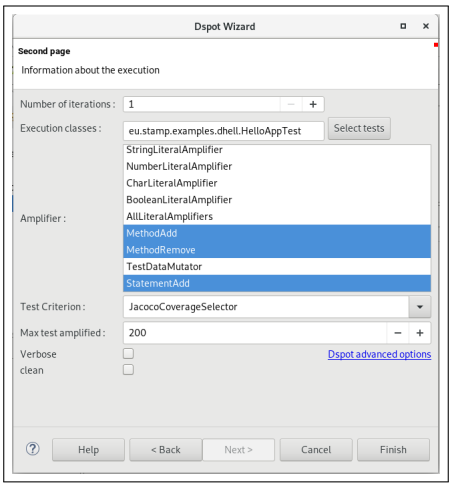


Figure 5.9: Screenshot of STAMP IDE showing an execution of DSpot configuration wizard

Test cases https://github.com/STAMP-project/dspot/blob/master/dspot/src/test/java/eu/stamp_project/automaticbuilder/GradleAutomaticBuilderTest.java and https://github.com/STAMP-project/dspot/blob/master/dspot/src/test/java/eu/stamp_project/automaticbuilder/GradleAutomaticBuilderWithDescartes.java show how to instantiate the GradleAutomaticBuilder and pass all the needed parameters. Generally speaking the build system is passed as an execution parameter to DSpot. The AutomaticBuilderFactory instantiates it accordingly to this parameter, assuming as default value the MavenAutomat-

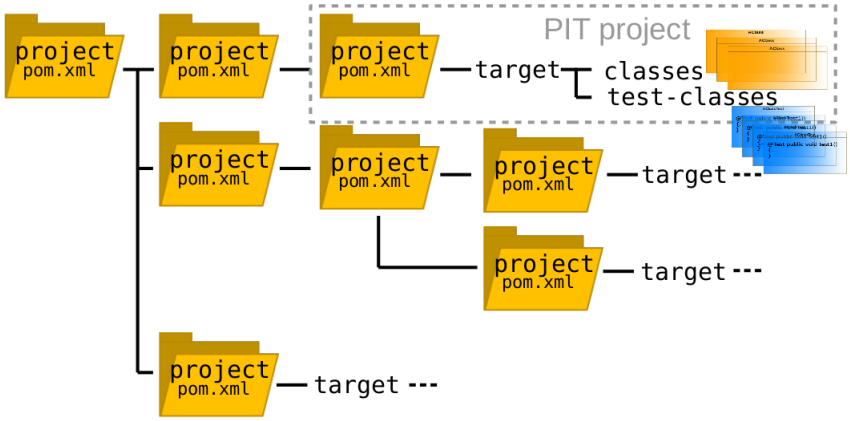


Figure 5.10: PIT project, i.e. PIT definition of a project. (test classes are blue and classes to be mutated are orange)

icBuilder (the Automtic BUilder parameter is optional). The Gradle Automatic Builder inject in the Gradle build file a specific amplification task in order to execute the amplified test cases. When the execution ends, the Gradle Automatic Builder reset the build file to the original status.

5.4.2 Development status

At the moment GradleAutomaticBuilder support is aligned with the latest DSpot version available. For a limit of the Gradle Tooling API (see https://github.com/gradle/gradle/issues/4215), it's not possibile to execute programmatically several tasks as retrieving the current Java project dependencies: this feature is available just for Gradle projects which use Eclipse or Intellij (these tools provide additional information about the project classpath). Plain Java projects lack this feature, so the only way to execute programmatically specific tasks is modifying on-the-fly the build file injecting the required tasks, and then reset the original build file at the end of the execution.

5.4.3 Future work

- Keep the code aligned with DSpot evolution
- More investigation about leveraging Gradle Tooling API in order to support also plain Gradle projects (a custom task implementation is needed)
- Investigation about the possibility to support Gradle multi-module projects

5.5 PitMP

In the context of the development of Descartes we faced an issue with PITest: PIT does not manage multi-module projects. Actually the PITest community asked for this feature in PITest, but it is still not supported.

5.5.1 The multi-module project issue

Figure 5.10 shows what is a project for PITest, let us call it a module here. It means that only the classes in this module will be mutated by PIT. If the tests of a module use classes of another module, those classes will not be mutated by PITest. Yet, it is common for projects to separate (unit) tests from the source code in the project tree. So in extreme cases you will have zero mutation in such project.

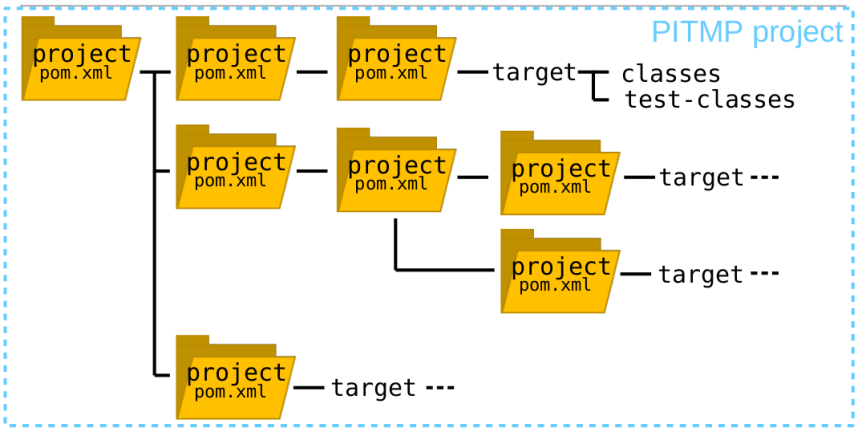
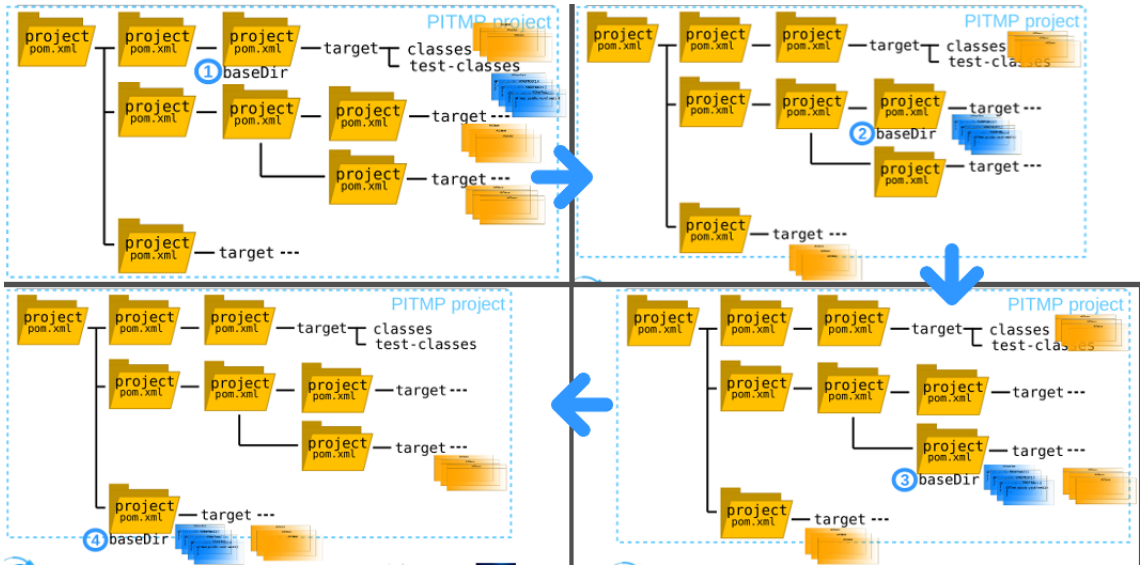


Figure 5.11: PitMP project, i.e. PitMP definition of a project.

Figure 5.12: PitMP execution on a multi-module project (test classes are blue and classes to be mutated are orange).



Such structure of tests is also used in STAMP use cases, which group tests into several modules and so requires mutation analysis through separate modules.

5.5.2 What PitMP does?

PitMP a Maven plugin able to run PIT on multi-module projects. The figure 5.11 shows what is a project for PitMP. If we keep the same definition of module, that is what we call a multi-module project. PitMP runs the test suite as PIT does, just extending the list of classes to be mutated to the whole project tree, instead of mutating only the classes of the test suite module. The figure 5.12 shows all steps of a PitMP execution on a multi-module project..

5.5.3 How does it work?

PitMP extends PIT, it does not rewrite any feature. This choice of implementation allows to use all the PITest features, including future improvements. Of course it means we have to follow up PITest

releases and to synchronize our releases with PITest releases, but it is still much less development than forking PITest or developing the same features.

PitMP extends PIT, so all the properties of PITtest can be used. We only need 2 PitMP specific properties: Running PitMP from a module directory will NOT work.

- targetModules: to run PIT only on specified modules
- skippedModules: to run PIT only on specified modules

5.5.4 Maven plugin for Descartes

Since Descartes is a mutation engine for PITest, we added a specific goal to our PitMP plugin to run Descartes automatically, instead of having to configure it in the pom.xml file of a project. This goal also facilitates the implementation of Descartes as a micro service.

5.5.5 Development status

From the first release v1.0.1 in January 2018 to the moment in which this document is being written, the Github repository of PitMP shows 103 additional commits to the master branch. These changes have been mostly directed to: 1) fix several bugs and issues reported in the tracker, 13 since the beginning of the project); 2) keep the code up-to-date with the regular releases and API changes of PITest; and 3) adding new features (Released in Maven Central, Descartes goal, JUnit5 support) to make it more useful and easier to adopt by developers.

5.5.6 Future work

- Combine results of test suites, I.e. aggregate Descartes reports from multiple maven projects.
- Optimize PitMP execution by excluding detected mutations: Right now PitMP may analyze the same mutation several times if it is covered by tests in different modules. So, before running the tests on a module it would be nice to check if the mutation was detected before by the tests of a previous module.
- Check Class Path conflicts: Adding paths into the class path of all depending modules is done without checking if 2 versions of the same component (jar file) are used. It would be useful to detect this kind of conflict: should it be an error or a warning ? Is there use cases which need to use 2 versions of the same component ?
- Add JUnit tests: There is automated tests but not with JUnit.

5.6 DSpot and Descartes microservices

When computing resources aren't available for test amplification, a possible solution to overcome this issue is to have test amplification as remote services. A microservice architecture fits very well for this purpose, and it was taken into account since the beginning of the project. Having this microservices allows for making it available STAMP in SaaS way.

5.6.1 How does it work?

Considering DSpot and Descartes, two possible scenarios are the most interesting:

- test amplification on current developer working copy
- test amplification on a specific branch within the code repository

The first scenario requires that the developer, in order to use DSpot and Descartes as a service, has to upload source code from her working copy within her working station. The second scenario requires that the developer has to provide remote DSpot/Descartes services with the reference to the specific branch. Invoking the amplification services is performed in two steps:

- upload the source code (HTTP POST method)
- get amplified test case (HTTP GET method)

Figure 5.13: Screenshot of SwaggerHub project hosting release 0.1.0 and 0.2.0 of STAMP microservices API

5.6.2 Development status

At the moment APIs were designed to support the first scenario (upload the working copy). Descartes can be used as a possible mutator. SwaggerHub was used for this purpose and a cuouple of versions are available at https://app.swaggerhub.com/apis/stamp-project/stamp-api/0.2.0. No implementation is provided yet. An investigation was conducted to select a microservices development framework and the choice was made in favour of Seedstack (http://seedstack.org/), a project belonging to OW2 ecosystem.

5.6.3 Future work

- Design APIs to execute Descartes as a standalone service
- Design APIs for the second scenario (upload source code from code repository)
- Implement APIs as microservices and make them available as Docker images
- Define microservices deployment scenarios

Chapter 6

Conclusion

This deliverable reports on the development activities related to the unit test amplification tool box. We develop to main tools for test assessment and amplification. For each of these tools, this report has clarified the development made since D1.2, as well as the current experiments and results:

- DSpot, a tool to automatically amplify JUnit test cases (cf. chapter 2).
- Descartes, a tool to locate pseudo-tested methods (cf. chapter 3)

A key novelty in this report, compared to D1.2, is the addition of a novel set of tools related to the integration of amplification in continuous integration pipelines (cf. chapter 4, chapter 5). These tools consolidate the core tools of Task 1.2 and Task 1.3, and they also fulfill the requirements of industrialization of Task 1.4.

Bibliography

[1] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Program mutation: A new approach to program testing. *Infotech State of the Art Report, Software Testing*, 2(1979):107–126, 1979.

[2] R. Niedermayr, E. Juergens, and S. Wagner. Will my tests tell me if I break this code? In *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*, pages 23–29, New York, NY, USA, 2016. ACM Press.

[3] Y. S. and H. M. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120.

[4] O. L. Vera-Pérez, B. Danglot, M. Monperrus, and B. Baudry. A comprehensive study of pseudo-tested methods. *CoRR*, abs/1807.05030, 2018.

[5] Q. Zhu, A. Panichella, and A. Zaidman. A systematic literature review of how mutation testing supports quality assurance processes. *Software Testing, Verification and Reliability*, page e1675.