STAMP

Deliverable D1.5

Final report about the amplification process for unit test suites

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

Deliverable Number	:	D1.5
Title of Deliverable	:	Final report about the amplification process for unit test suites
Dissemination Level	:	Public
Version	:	1.0
Latest version	:	https://github.com/STAMP-project/docs-forum/blob/master/docs/d15_final_report_unit_test_amplification.pdf
Contractual Delivery Date	:	M36 November, 30 2019
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)	:	Pierre-Yves Gibello, OW2 Assad Montasser, OW2

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	Oscar Luis Vera Perez, INRIA
1.2	DSpot	Benjamin Danglot, INRIA
1.3	Intro and global edit	Benoit Baudry, KTH
1.5	Reviewers, typos	Caroline Landry, INRIA

Contents

Chapter 1

Introduction

This deliverable is the last one for WP1 of STAMP, focusing on the two core tools that were developed for this workpackage: Descartes (chapter 2) and DSpot (chapter 3). The tools have been developed, maintained and extensively experimented throughout the whole project. This final deliverable focuses on documenting the tools from a user and a contributor perspective.

This tool-box supported the exploration of novel research questions and at provided relevant feedback to developers to improve the quality of industrial open source projects. The core research question we addressed within workpackage 1 was 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?

Chapter 2

Descartes

Descartes evaluates the capability of a test suite using extreme mutation testing [3]. It has been conceived as an extension of PITest [1]. In the following sections we describe how does it work, how it can be used and how can other developers contribute to the project.

2.1 How does it work?

Mutation testing, proposed in 1978 by DeMillo and colleagues [2], is a technique to verify if a test suites can detect possible bugs. Mutation testing does it by introducing small changes or faults into the original program. These modified versions are called *mutants*. A good test suite should able to *kill* or detect a mutant. Traditional mutation testing works at the instruction level, *e.g.*, , replacing > by <=, so the number of generated mutants can be huge. This makes the analysis time consuming, as every mutant should be challenge against every test case in the test suite that could execute its code.

Niedermayr and colleagues proposed *extreme mutation* that eliminate at once the whole logic of a method under test. If the method is `void`, then all instructions in the body are removed. Otherwise, the body is replaced by a single return instruction with a predefined constant value. Extreme mutation can be used to detect *pseudo-tested* methods, that is, those methods whose body can be removed and their results can be altered by any value and yet the test suite does not notice these changes.

Descartes extends PITest with an effective implementation of extreme mutation and automatically finds pseudo-tested methods in a Java project.

Mutation operators

Descartes is able to analyze any method using the following mutation operators:

`void` mutation operator

This operator accepts a `void` method and removes all the instructions on its body in the way it is shown in Listing 2.1

Listing 2.1: `void` mutation operator

```
class A {

  int field = 3;

  public void Method(int inc) {
    field += 3;
  }
```

```
}

// Mutant

class A {

  int field = 3;

  public void Method(int inc) { } // Code was removed

}
```

`null` mutation operator

This operator accepts a method returning a reference return type and replaces all instructions with `return null` as shown in Listing 2.2.

Listing 2.2: `null` mutation operator

```
class A {
    public A clone() {
        return new A();
    }
}

// Mutant

class A {
    public A clone() {
        return null;
    }
}
```

`empty` mutation operator

This operator targets methods that return arrays. It replaces the entire body with a `return` statement that produces an empty array of the corresponding type. An example is shown in Listing 2.3.

Listing 2.3: `empty` mutation operator

```
class A {
  public int[] getRange(int count) {
    int[] result = new int[count];
    for(int i=0; i < count; i++) {
      result[i] = i;
    }
    return result;
  }
}

// Mutant

class A {
  public int[] getRange(int count) {
    return new int[0];
```

```
  }
}
```

Constant mutation operator

This operator accepts any method with primitive or `String` return type. It replaces the method body with a single instruction returning a predefined constant. Listing 2.4, shows the effects of the operator when `3` is specified. This mutation operator can be configured with any Java literal value.

Listing 2.4: `constant` mutation operator

```
class A {
    int field;

    public int getAbsField() {
        if(field >= 0)
            return field;
        return -field;
    }
}

// Mutant

class A {
    int field;

    public int getAbsField() {
        return 3;
    }
}
```

`new` mutation operator

This operator accepts any method whose return type has a constructor with no parameters and belongs to a `java` package. It replaces the code of the method by a single instruction returning a new instance. Listing 2.5 shows an example where a method returning `ArrayList` is transformed.

Listing 2.5: `new` mutation operator

```
class A {
    int field;

    public ArrayList range(int end) {
        ArrayList l = new ArrayList();
        for(int i = 0; i < size; i++) {
            A a = new A();
            a.field = i;
            l.add(a);
        }
        return l;
    }
}

// Mutant

class A {
```

Table 2.1: Replacements done by the `new` mutation operator

Return Type	Replacement
Collection	ArrayList
Iterable	ArrayList
List	ArrayList
Queue	LinkedList
Set	HashSet
Map	HashMap

```
    int field;

    public List range(int end) {
        return new ArrayList();
    }
}
```

For specific classes, the mutation operator returns an instance of a known derived class. The list of replacements is shown in Table 2.1.

This means that if a is supposed to return an instance of `Collection` the code of the mutated method will be `return new ArrayList();`.

This operator is not enabled by default.

`optional` mutation operator

This operator accepts any method whose return type is `java.util.Optional`. It replaces the code of the method by a single instruction returning an *empty* instance. Listing 2.6 shows an example. This operator is not enabled by default.

Listing 2.6: `optional` mutation operator

```
class A {
    int field;

    public Optional<Integer> getOptional() {
        return Optional.of(field);
    }
}

// Mutant

class A {
    int field;

   public Optional<Integer> getOptional() {
           return Optional.empty();
    }
}
```

2.2 User Guide

Descartes requires PITest to be able to work. To check how to use PITest users may consult PITest's official documentation [1]. Descartes is able to analyze Java, Kotlin and Scala projects built with Maven, Gradle, Ant or even from the command line.

Running Descartes on a Maven project

Descartes can be used to target a Maven project a user can opt to configure the `pom.xml` file of the project, or use PitMP from the command line.

Listing 2.11 shows the minimal configuration to add to a `pom.xml` to used Descartes. Once this configuration is added to the `pom.xml` Descartes can be executed from the command line as shown in Listing 2.9. This is the same goal that can be used to execute PITest. The configuration in the `pom.xml` declares Descartes as a dependency and activates the right mutation engine.

Listing 2.7: Maven configuration to use Descartes

```
<build>
  ...
  <plugins>
    ...
    <plugin>
      <groupId>org.pitest</groupId>
      <artifactId>pitest-maven</artifactId>
      <version>1.4.10</version>
      <configuration>
        <mutationEngine>descartes</mutationEngine> <!-- Selecting Descartes as the
            active mutation engine -->
      </configuration>
      <dependencies>
        <!-- Descartes dependency -->
        <dependency>
          <groupId>eu.stamp-project</groupId>
          <artifactId>descartes</artifactId>
          <version>1.2.6</version>
        </dependency>
      </dependencies>
    </plugin>
    ...
  </plugins>
  ...
</build>
```

Listing 2.8: Maven goal to launch descartes Descartes

```
mvn clean test org.pitest:pitest-maven:mutationCoverage
```

By using PitMP there is no need to modify the pom.xml file. **??** shows the command line to execute Descartes. This command has to be executed in the root folder of the project.

Listing 2.9: Maven goal to launch descartes Descartes

```
mvn clean test org.stamp-project:pitmp-maven-plugin:descartes
```

[1] http://pitest.org

Configuring mutation operators

It is possible to change the default selection of mutation operators. They can be specified in the `pom.xml` file. **??** shows and example where the `void` operator is used together with constant operators to make the methods return `4`, `"some string"` and `false`.

Listing 2.10: Maven configuration to use Descartes

```
<build>
  ...
  <plugins>
    ...
    <plugin>
      <groupId>org.pitest</groupId>
      <artifactId>pitest-maven</artifactId>
      <version>1.4.10</version>
      <configuration>
        <mutationEngine>descartes</mutationEngine>
        <!-- Configuring operators -->
        <mutators>
          <mutator>void</mutator>
          <mutator>4</mutator>
          <mutator>"some string"</mutator>
          <mutator>false</mutator>
        </mutators>
      </configuration>
        <dependencies>
        <dependency>
            <groupId>eu.stamp-project</groupId>
            <artifactId>descartes</artifactId>
            <version>1.2.6</version>
        </dependency>
        </dependencies>
    </plugin>
    ...
    </plugins>
    ...
</build>
```

Possible values for the content of a `mutator` item are: `void`, `empty`, `optional`, `new`, `null` and any Java literal as stated in the language specification[2]. Table 2.2 shows examples of literal values than can be used. The use of negative values, as well as binary, octal and hexadecimal literals is allowed. To specify `short` and `byte` constants it is possible to use a cast-like notation. A constant operator will transform only those methods whose return type is the same as the specified literal.

Listing 2.11 shows the configuration used by Descartes when no operator is configured.

Listing 2.11: Default configuration

```
<mutators>
  <mutator>void</mutator>
  <mutator>null</mutator>
  <mutator>true</mutator>
  <mutator>false</mutator>
  <mutator>empty</mutator>
  <mutator>0</mutator>
  <mutator>1</mutator>
```

[2] https://docs.oracle.com/javase/specs/jls/se7/html/jls-3.html#jls-3.10

Table 2.2: Examples of literal values that can be used to specify a constant mutation operator

Literal	Type
`true`	`boolean`
`1`	`int`
`2L`	`long`
`3.0f`	`float`
`-4.0`	`double`
`'a'`	`char`
`"literal"`	`string`
`(short) -1`	`short`
`(byte)0x1A`	`byte`

```
  <mutator>(byte)0</mutator>
  <mutator>(byte)1</mutator>
  <mutator>(short)0</mutator>
  <mutator>(short)1</mutator>
  <mutator>0L</mutator>
  <mutator>1L</mutator>
  <mutator>0.0</mutator>
  <mutator>1.0</mutator>
  <mutator>0.0f</mutator>
  <mutator>1.0f</mutator>
  <mutator>'\40'</mutator>
  <mutator>'A'</mutator>
  <mutator>""</mutator>
  <mutator>"A"</mutator>
</mutators>
```

Stop Methods

Descartes avoids some methods that are generally not interesting and may introduce false positives such as simple getters, simple setters, empty void methods or methods returning constant values, delegation patterns as well as deprecated and compiler generated methods. Those methods are automatically detected by inspecting their code. The exclusion of stop methods can be configured.

For this, Descartes includes a PITest feature [3] named `STOP_METHODS`. This feature is enabled by default. The feature parameter `exclude` can be used to prevent certain methods to be treated as stop methods and bring them back to the analysis. Table 2.3 shows all possible values of this parameter and their meaning.

[3] http://pitest.org/quickstart/advanced/#mutation-interceptor

Table 2.3: Examples of literal values that can be used to specify a constant mutation operator

exclude	Method description	Example
empty	void methods with no instruction.	`public void m(){}`
enum	Methods generated by the compiler to support enum types (values and valueOf).	
to_string	toString methods.	
hash_code	hashCode methods.	
deprecated	Methods annotated with @Deprecated or belonging to a class with the same annotation.	`@Deprecated public void m() {...}`
synthetic	Methods generated by the compiler.	
getter	Simple getters.	`public int getAge() { return this.age; }`
setter	Simple setters. Includes also fluent simple setters.	`public void setX(int x) { this.x = x; }` `public A setX(int x){ this.x = x; return this; }`
constant	Methods returning a literal constant.	`public double getPI() { return 3.14; }`
delegate	Methods implementing simple delegation.	`public int sum(int[] a, int i, int j) {return this.adder(a, i, j); }`
clinit	Static class initializers.	
return_this	Methods that only return this.	`public A m() { return this; }`
return_param	Methods that only return the value of a real parameter	`public int m(int x, int y) { return y; }`
kotlin_setter	Setters generated for data classes in Kotlin	

Listing 2.14 shows how to configure the `pom.xml` to make Descartes analyze deprecated methods. The `features` element should be added inside the `configuration` element in the plugin configuration. Listing 2.13 shows how to enable at the same time enum and `toString` methods.

Listing 2.12: Analyzing deprecated methods

```
<features>
  <feature>
  <!-- This will allow descartes to mutate deprecated methods -->
    +STOP_METHODS(except[deprecated])
  </feature>
</features>
```

Listing 2.13: Analyzing enum and `toString` methods

```
<features>
  <feature>
  <!-- This will allow descartes to mutate toString and enum generated methods -->
    +STOP_METHODS(except[to_string] except[enum])
  </feature>
</features>
```

It is possible to disable this feature completely. **??** shows how.

Listing 2.14: Disabling stop methods

```
<features>
  <feature>
  <!--No method is considered as a stop method and therefore all of them will be
      mutated -->
    -STOP_METHODS()
  </feature>
</features>
```

Descartes also includes another feature to avoid mutating methods annotated with `@NotNull` using the `null` mutation operator. This feature is enabled by default and can be disabled as shown in Listing 2.15.

Listing 2.15: Enabling the mutation of methods annotated with `@NotNull` with the `null` mutation operator

```
<features>
  <feature>
    -AVOID_NULL()
  </feature>
</features>
```

Configuring the output

All PITest reporting extensions work with Descartes. They can produce output in `XML`, `CSV` and `HTML` formats. The `HTML` format also includes a code coverage report.

Descartes also provides three new reporting extensions:

- a general reporting extension supporting `JSON` files. It works also with the default mutation engine for PITest.
- a `METHODS` reporting extension designed that generates a `JSON` file with information about pseudo- and partially-tested methods. A method is said to be pseudo-tested if all mutants inside its body survive and partially-tested if some mutants survive the analysis and other were detected at the same time.

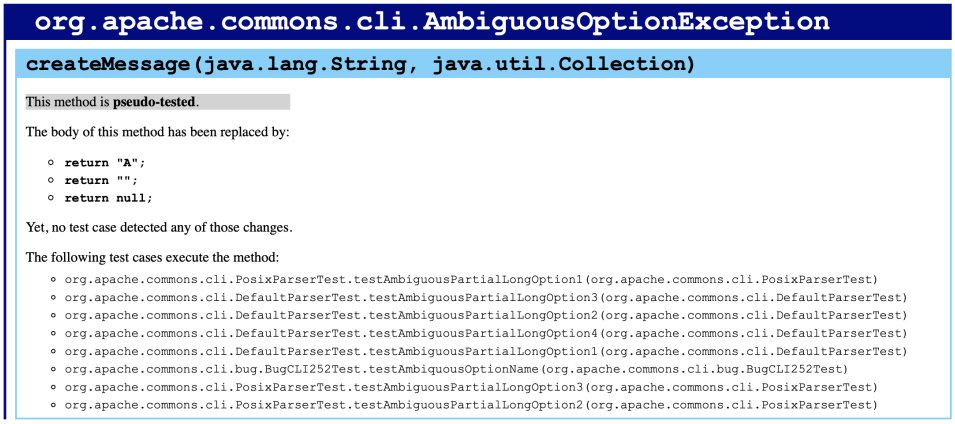

Figure 2.1: Example of a report generated by Descartes

- an `ISSUES` reporting extension that is a human readable version of `METHODS`. An example can be seeing in Figure 2.1

Listing 2.16 shows how to configure all the reporting extensions for Descartes.

Listing 2.16: Configuring the reporting extensions

```
<plugin>
  <groupId>org.pitest</groupId>
  <artifactId>pitest-maven</artifactId>
  <version>1.4.10</version>
  <configuration>
    <outputFormats>
      <value>JSON</value>
      <value>METHODS</value>
      <value>ISSUES</value>
    </outputFormats>
    <mutationEngine>descartes</mutationEngine>
  </configuration>
    <dependencies>
    <dependency>
      <groupId>eu.stamp-project</groupId>
      <artifactId>descartes</artifactId>
      <version>1.2.6</version>
    </dependency>
  </dependencies>
</plugin>
```

Running Descartes on a Gradle project

To run Descartes in a Gradle project it is required to first follow the instructions to set up PITest. This instructions can be found in the following URL: http://gradle-pitest-plugin.solidsoft.info/.

Descartes must be declared in the `dependencies` block i.e. add `pitest 'eu.stamp-project:descarte` in the aforementioned block. Then, the mutation engine and all other configurations can be added in a `pitest` block. Listing 2.17 shows a full example.

Listing 2.17: Gradle configuration

```
buildscript {
  repositories {
    mavenCentral()
    mavenLocal()
  }

  configurations.maybeCreate("pitest")

  dependencies {
    classpath 'info.solidsoft.gradle.pitest:gradle-pitest-plugin:1.4.0'
    pitest 'eu.stamp-project:descartes:1.2.6'
  }
}

apply plugin: "info.solidsoft.pitest"

pitest {
  targetClasses = ['my.package.*'] //Assuming all classes in the project are
      located in the my.package package.
  mutationEngine = "descartes"
  pitestVersion = "1.4.10"
}
`
```

Is it important to notice that the `pitestVersion` property must be specified to avoid version issues with the default version shipped with the Gradle PITest plugin.

Then, to run the tool a user has to execute the command `gradle pitest` in the root folder of the project.

2.3 Contributor Guide

The functionalities of Descartes can be divided into three main groups:

- the mutation engine and mutation operators
- features to filter out unproductive or trivial mutations
- reporting extensions and method classification.

All these functionalities are implemented using PITest's extension mechanisms. In this way we leverage other already mature features such as project and configuration handling and test execution.

Figure 2.2 shows the main classes involved in the implementation of the mutation engine.

`DescartesMutationEngine` handles both Descartes and PITest configurations. It manages the mutation operators, filters and restrictions that the user has enabled so it can decide for a given method the set of mutation operators to use. `DescartesMutater` handles the discovery and creation of mutants for each class. `MutationPointFinder` finds all mutants with the help of `DescartesMutationEngine`, it is implemented as an ASM `ClassVisitor`[4]. `DescartesMutationEngine` and `DescartesMutater` implement the `MutationEngine` and `Mutater` interfaces respectively. Both interfaces are provided by PITest.

All mutation operators in Descartes are derived from `MutationOperator` as shown in Figure 2.3. They should implement two methods. `canMutate` is used to know if a given method can be mutated according to the logic of the operator and the signature of the method. For example, the `nuemptyll` mutation operator would only mutate methods returning an array. `generateCode` generates the actual program variant or mutant.

[4] https://asm.ow2.io/javadoc/org/objectweb/asm/ClassVisitor.html

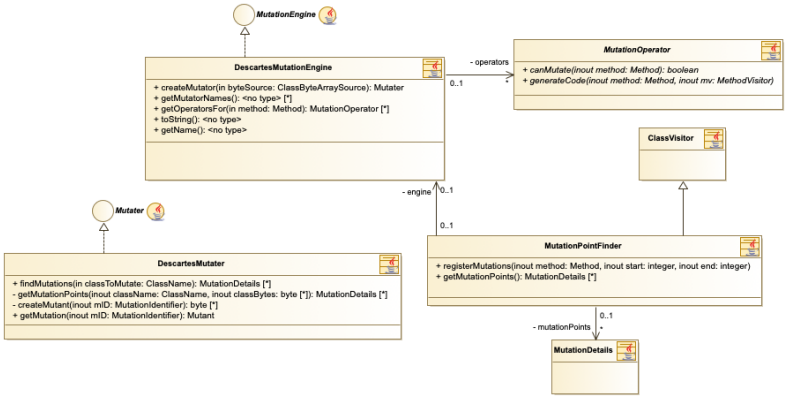

Figure 2.2: Mutation engine architecture

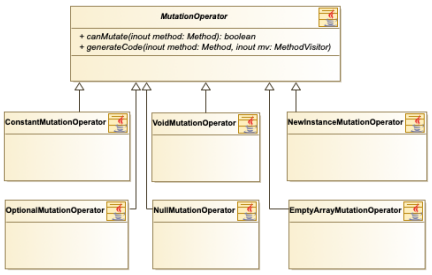

Figure 2.3: Example of a report generated by Descartes

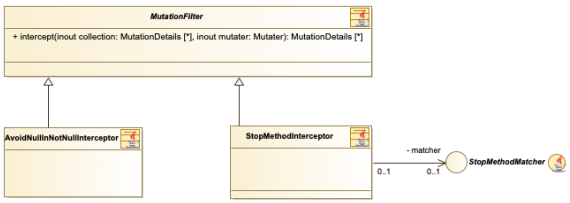

Figure 2.4: Example of a report generated by Descartes

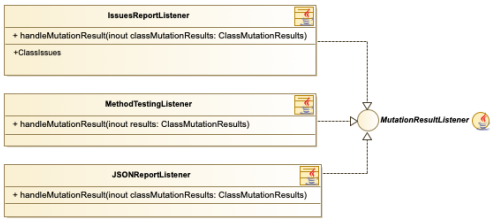

Figure 2.5: Example of a report generated by Descartes

Descartes provides `StopMethodInterceptor` and `AvoidNullInNotNullInterceptor` as filters to avoid analyzing mutations that are trivial or not helpful. The former skips mutations made in methods that are generally not targeted in tests such as simple getters or setters. The latter skips `null` mutations in methods annotated with `@NotNull`. Both classes extend the `MutationFilter` class from PITest.

Descartes provides its own reporting extensions: `JSONReportListener`, `MethodTestingListener` and`IssuessReportListener` shown in Figure 2.5. These three classes implement the `MutationResultListener` interface from PITest.

All classes implementing interfaces or extending classes from PITest require a `Factory` counterpart as per the PITest architecture. For example, `DescartesMutationEngine` requires a `DescartesMutationEngineFactory`. These factories should create instances of the intended classes. As such an instance of `DescartesMutationEngineFactory` should create an instance of `DescartesMutationEngine`.

Installing and building from source

Listing 2.18 shows the steps to build Descartes from source. It first clones the repository and then builds the project.

Listing 2.18: Steps to build Descartes

```
git clone https://github.com/STAMP-project/pitest-descartes.git
cd  pitest-descartes
mvn install
```

The `master` branch is the main branch for development. Releases are tagged in that same branch. Contributors are encouraged to create new branches for modifications and new features and then create a pull request to the `master` branch in the Github repository.

Chapter 3

DSpot: tool for unit test amplification

DSpot amplifies unit test suites according to a specific engineering goal (e.g., improve the test suite's mutation score). In the following sections we describe the internal flow of the tool, how it can be used and how can other developers contribute to the project.

3.1 User Guide

3.1.1 Prerequisites

You need Java and Maven.

DSpot uses the environment variable MAVEN_HOME, ensure that this variable points to your maven installation. Example:

```
export MAVEN_HOME=path/to/maven/
```

DSpot uses maven to compile, and build the classpath of your project. The environment variable JAVA_HOME must point to a valid JDK installation (and not a JRE).

3.1.2 Releases

We advise you to start by downloading the latest release, see https://github.com/STAMP-project/dspot/releases.

3.1.3 First Tutorial

After having downloaded DSpot (see the previous section), you can run the provided example by running `eu.stamp_project.Main` from your IDE, or with

“‘ java -jar target/dspot-LATEST-jar-with-dependencies.jar –example “‘

replacing ‘LATEST‘ by the latest version of DSpot, *e.g.*, 2.2.1 would give : $dspot - 2.2.1 - jar - with - dependencies.jar$

This example is an implementation of the function chartAt(s, i) (in src/test/resources/test-projects/), which returns the char at the index i in the String s.

In this example, DSpot amplifies the tests of chartAt(s, i) with the FastLiteralAmplifier, which modifies literals inside the test and the generation of assertions.

The result of the amplification of charAt consists of 6 new tests, as shown in the output below. These new tests are written to the output folder specified by configuration property ‘outputDirectory‘ (./target/dspot/output/).

```
Initial instruction coverage: 30 / 34
88.24%
Amplification results with 5 amplified tests.
Amplified instruction coverage: 34 / 34
100.00%
```

3.1.4 Command Line Usage

You can then execute DSpot by using:

```
java -jar /path/to/dspot-LATEST-jar-with-dependencies.jar --absolute-path-to-pr
```

Amplify a specific test class

```
java -jar /path/to/dspot-*-jar-with-dependencies.jar eu.stamp_project.Main --ab
```

Amplify specific test classes according to a regex

```
java -jar /path/to/dspot-LATEST-jar-with-dependencies.jar --absolute-path-to-pr
java -jar /path/to/dspot-LATEST-jar-with-dependencies.jar --absolute-path-to-pr
```

Amplify a specific test method from a specific test class

```
java -jar /path/to/dspot-LATEST-jar-with-dependencies.jar --absolute-path-to-pr
```

3.1.5 Command Line Options

We list here a small set of the command line options. An exhaustive list is available on the github repository: https://github.com/STAMP-project/dspot.git.

- –absolute-path-to-project-root=<absolutePathToProjectRoot> Specify the path to the root of the project. This path must be absolute.
- -a, –amplifiers=<amplifiers>[,<amplifiers>...] Specify the list of amplifiers to use. By default, DSpot does not use any amplifiers (None) and applies only assertion amplification.
- -s, –test-selector, –test-criterion=<selector> Specify the test adequacy criterion to be maximized with amplification.

3.1.6 Maven plugin usage

You can execute DSpot using the maven plugin. This plugin let you integrate unit test amplification in your Maven build process. You can use this plugin on the command line as the jar:

```
# this amplifies the Junit tests to kill more mutants
mvn eu.stamp-project:dspot-maven:amplify-unit-tests

# this amplifies the Junit tests to improve coverage
mvn eu.stamp-project:dspot-maven:amplify-unit-tests -Dtest-criterion=JacocoCove
```

All the option can be pass through command line by prefixing the option with -D. For example:

```
mvn eu.stamp-project:dspot-maven:amplify-unit-tests -Dtest=my.package.TestClass
```

or, you can add the following to your *pom.xml*, in the plugins section of the build:

```
<plugin>
<groupId>eu.stamp-project</groupId>
<artifactId>dspot-maven</artifactId>
<version>LATEST</version>
<configuration>
<!-- your configuration -->
</configuration>
</plugin>
```

Each command line options is translated into an option for the maven plugin. You must prefix each of them with '-D'. Examples:

* '–test my.package.MyTestClass1:my.package.MyTestClass2' gives '-Dtest=my.package.MyTestClass1,my.packag
* '–output-path output' gives '-Doutput-path=output'

3.2 Contributor Guide

DSpot's developer team encourages contributions in form of pull requests. Pull requests must include tests that specify the changes.

For each pull request opened, Travis CI is triggered. Our CI contains different jobs that must all pass.

There are jobs that execute the test for the different module of DSpot: DSpot Core, DSpot Maven plugin, DSpot diff test selection, and DSpot prettifier.

There are also jobs for different kind of execution: from command line, using the maven plugin from command line and from a configuration in the pom, on large and complex code base.

We use a checkstyle to ensure a minimal code readability.

The code coverage (instruction level) must not decrease by 1% and under 80%.

Chapter 4

Conclusion

Over the three years of the STAMP project, the work in WP1 was articulated around the development of two core pieces of technology: Descartes and DSpot. The novel features, the scientific and technical investigations with these tools and their relation to state of the art have been discussed in deliverables D1.2, D1.3 and D1.4. This last deliverable for WP1 focused on user and contributors guides.

4.1 Publications for WP1, Oct-Nov 2019

Accepted for publication in November 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

Bibliography

[1] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque. PIT: A Practical Mutation Testing Tool for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 449–452. ACM.

[2] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. 11(4):34–41.

[3] 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. ACM Press.