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	:	1.8
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)	:	Benoit Baudry, KTH Benjamin Danglot, INRIA Caroline Landry, INRIA Martin Monperrus, KTH 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
1.6	Consolidate guides and edition	Benjamin Danglot, Oscar Luis Vera Perez, INRIA
1.7	Edit wrt review from OW2	Benoit Baudry, KTH
1.8	Final review: fixing references and typos	Benjamin Danglot, Caroline Landry, Oscar Luis Vera Perez, 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 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 it works, how to use it and how developers can 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 suite 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 be 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 challenged against every test case in the test suite that could execute its code.

Niedermayr and colleagues proposed *extreme mutation* that eliminates 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

`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 2.1: `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

}
```

Listing 2.2: `null` mutation operator

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

// Mutant

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

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];
  }
}
```

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

`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.

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.

`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.

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 method 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.

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;
    }
}
```

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 {
    int field;

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

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();
    }
}
```

`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.

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 target Maven projects. Users may opt to configure the `pom.xml` file of the project, or use PitMP from the command line.

Listing 2.7 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 2.8. 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 2.8: Maven goal to launch Descartes

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

By using PitMP there is no need to modify the pom.xml file. Listing 2.8 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

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

Configuring mutation operators

It is possible to change the default selection of mutation operators. They can be specified in the `pom.xml` file. Listing 2.10 shows an example where the `void` operator is used together with constant operators to make the methods return `4`, `"some string"` and `false`.

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

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.

[2] https://docs.oracle.com/javase/specs/jls/se7/html/jls-3.html#jls-3.10
[3] http://pitest.org/quickstart/advanced/#mutation-interceptor

Listing 2.10: Configuring a custom set of mutation operators

```
<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>
```

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`

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>
  <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>
```

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 2.12: 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>
```

Listing 2.12 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.

It is possible to disable this feature completely. Listing 2.14 shows how.

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.

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 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 others were detected at the same time.

Listing 2.14: Disabling stop methods

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

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

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

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 seen in Figure 2.1

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

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. These instructions can be found at the following URL: http://gradle-pitest-plugin.solidsoft.info/.

Descartes must be declared in the `dependencies` block. i.e. add:
`pitest 'eu.stamp-project:descartes:1.2.6`
as one of the dependencies. Then, the mutation engine and all other configurations can be added in a `pitest` block. Listing 2.17 shows a full example.

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

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>
```

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"
}
```

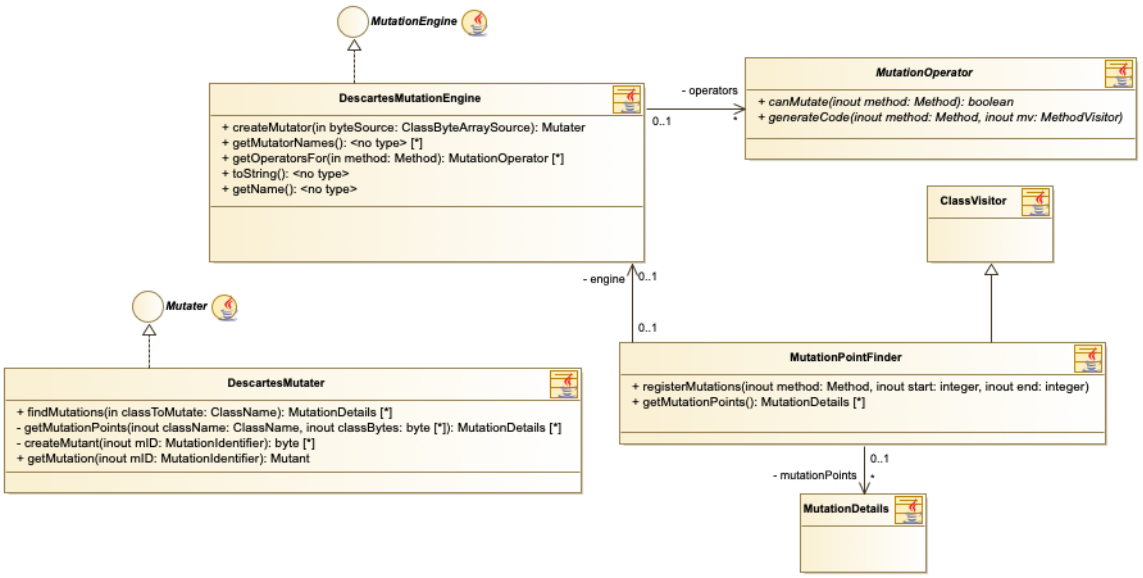

Figure 2.2: Mutation engine architecture

- 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 `empty` mutation operator would only mutate methods returning an array. `generateCode` generates the actual program variant or mutant.

PITest provides the `MutationFilter` interface. Classes implementing this interface can avoid the creation of certain mutants and even modify the mutation after, the mutation operators have identified all the mutation points. Descartes provides `StopMethodInterceptor` and `AvoidNullIn-NotNullInterceptor`, shown in Figure 2.4, 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` counter-

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

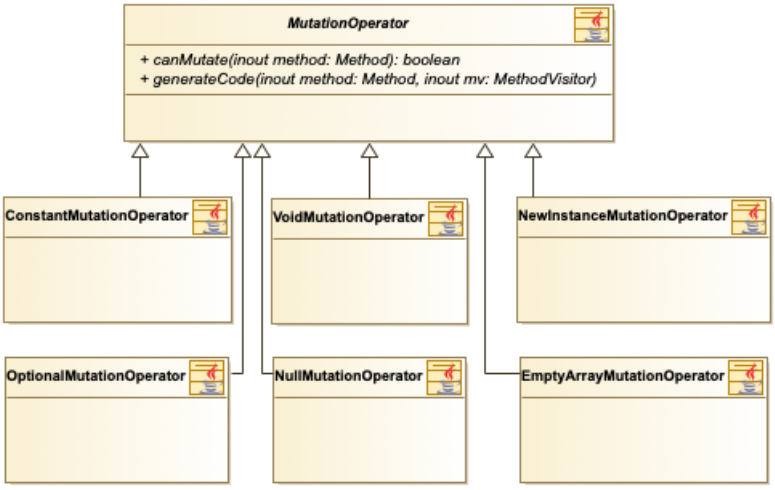

Figure 2.3: Mutation operators

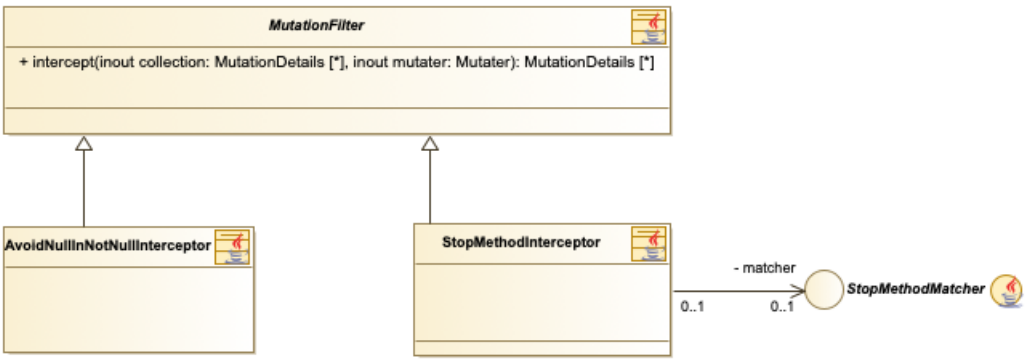

Figure 2.4: Mutation filters

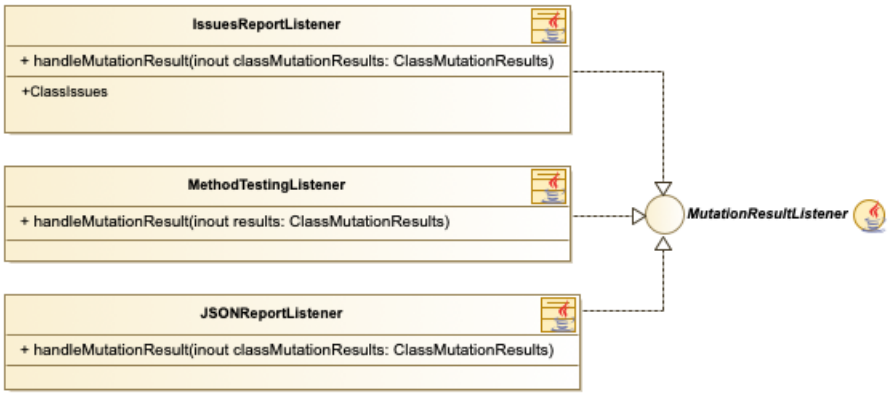

Figure 2.5: Example of a report generated by Descartes

Listing 2.18: Steps to build Descartes

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

Table 2.4: Development statistics

Number of tests	126
Number of commits	310
Number of branches	5
Number of contributors	12
Number of pull requests	12
Number of releases (Github)	14
Number of releases (Maven Central)	9
Number of LoC (Java)	4122

part 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.

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.

2.4 Development Stats

Table 2.4 presents some statistics from Descartes's project and development.

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 How does it work?

3.1.1 Principle

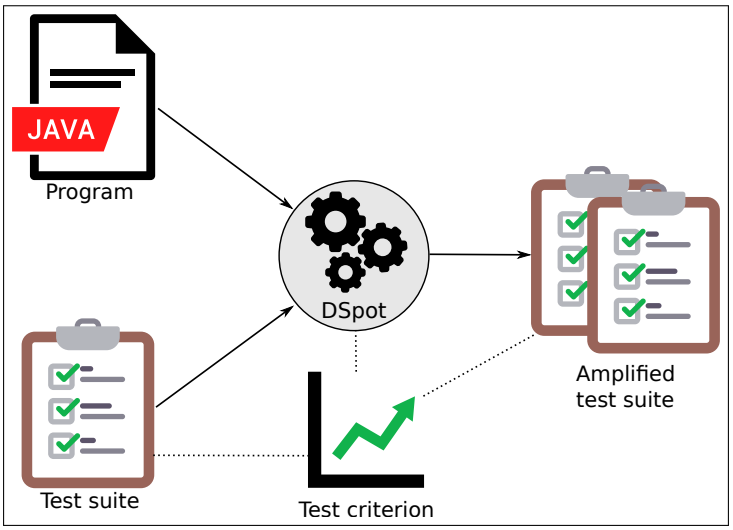

Figure 3.1: DSpot's principle

DSpot is a test amplification tool. Its goal is to improve an existing test suite according to a specific test-criterion.

As shown in figure Figure 3.1, DSpot takes as input the program, an existing test suite, and a test-criterion. The output of DSpot is a set of amplified test methods that are variants of existing test methods. When added to the existing test suite, it create an amplified test suite. This amplified test suite is better than the original test suite according to the test-criterion used during the amplification. For instance, one amplifies its test suite using branch coverage as test-criterion. This amplified test

Listing 3.1: Example of what DSpot produces: a diff to improve an existing test case.

```
@@ -144,7 +144,8 @@ public void testEmptyList() throws Exception
 ArrayList<Foo> foos = new ArrayList<Foo>();

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

suite will execute more branches than the exiting test suite, i.e. the one without amplified test methods. In DSpot there are for now 3 test-criterion available:

1) keeping amplified test methods that increase the mutation score;

2) keeping amplified test methods that increase the instruction coverage;

3) keeping amplified test methods that detect the behavioral difference between two versions of the same program.

3.1.2 Input & Output

DSpot's inputs are a program, a set of existing test methods and a test-criterion. The program is used as ground truth: in DSpot we consider the program used during the amplification correct. The existing test methods are used as a seed for the amplification. DSpot applies transformation individually to these test methods in order to improve the overall quality of the test suite with respect to the specified test-criterion.

DSpot produces variants of the test methods provided as input. These variants are called amplified test methods, since there are test methods that has been obtained using an amplification process. These amplified test methods are meant to be added to the test suite. By adding amplified test methods to the existing test suite, it creates an amplified test suite that improves the overall test suite quality. By construction, the amplified test suite is better than the original one with respect to the specified criterion.

An amplified test method's integration can be done in two way:

1) the developer integrates as it is the amplified test method into the test suite;

2) the developer integrate only the changes between the original test method and the amplified test method.

This enrich directly an existing test method.

Listing 3.1 shows an example of changes' set obtained using DSpot.

By construction, all DSpot's amplifications can be represented as a diff on an existing test method since amplified test methods are variants of existing ones.

3.1.3 Workflow

As shown in figure Figure 3.2, the main workflow of DSpot is composed of 3 main phases:

1) the modification of test code's inputs inspired by Tonella's technique [4], called "input space exploration"; this phase consists in modifying test values (*e.g.*, literals), objects and methods calls, the underlying details will be explained in subsection 3.2.1;

2) the addition of new assertions per Xie's technique [5], this phase is called "assertion improvement" The behavior of the system under test is considered as the oracle of the assertion, see subsection 3.2.2. In DSpot, the combination of both techniques, i.e. the combination of input space exploration and assertion improvement is called "test amplification";

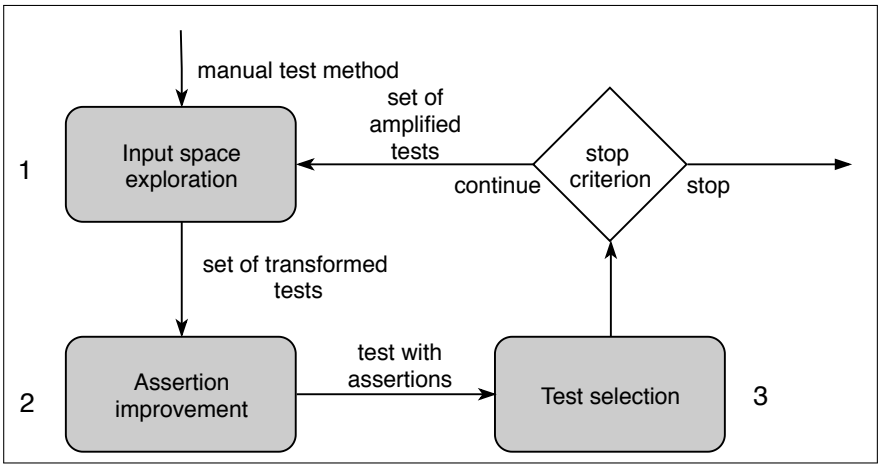

Figure 3.2: DSpot's workflow three main steps

Listing 3.2: An example of an object-oriented test case (inspired from Apache Commons Collections)

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

3) the amplified test methods selection according to a given test-criterion, *e.g.*, branch coverage. Eventually, DSpot either stops or continues to apply test amplification, according to a pre-defined stop-criterion. By doing this, DSpot stacks the transformation of test methods. In other words, DSpot amplifies already amplified test methods, which is possible because DSpot's output are real test methods.

In DSpot, the used stop-criterion is a number of iteration. However, one can imagine others kinds of stop-criterion such as a time budget, a test-criterion goal(*e.g.*, reach 50% of mutation score) or a finite number of amplified test methods.

3.1.4 Test Method Example

DSpot amplifies Java program's test methods, which are typically composed of two parts: test inputs and assertions.

Listing 3.2 illustrates an archetypal example of such a test case: first, from line 4 to line 6, the test input is created through a sequence of object creations and method calls; then, at line 8, the tested behavior is actually triggered; the last part of the test case at 11 and 12, the assertion part, specifies and checks the conformance of the observed behavior with the expected one. Note that this notion of call sequence and complex objects is different from test inputs consisting only of primitive values.

Table 3.1: Literal test transformations in DSpot

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

Best target test

By the algorithm's nature, unit tests (vs integration test) are the best target for DSpot. The reasons are behind the very nature of unit tests: First, they have a small scope, which allow DSpot to intensify its search while an integration test, that contains a lot of code, would make DSpot explore the neighborhood in different ways. Second, that is a consequence of the first, the unit tests are fast to be executed against integration test. Since DSpot needs to execute multiple times the tests under amplification, it means that DSpot would be executed faster when it amplifies unit tests than when it amplified integration tests.

3.2 Algorithm

3.2.1 Input Space Exploration Algorithm

DSpot aims at exploring the input space so as to set the program in new, never explored states. To do so, DSpot applies code transformations to the original manually-written test methods. ***I-Amplification*** for Input Amplification, is the process of automatically creating new test input points from existing test input points. DSpot uses three kinds of *I-Amplification*:

1) Amplification of literals: the new input point is obtained by changing a literal used in the test (numeric, boolean, string). These transformations are summarized in Table 3.1.

2) Amplification of method calls: DSpot manipulates method calls as follows: DSpot duplicates an existing method call; removes a method call; or adds a new invocation to an accessible method with an existing variable as target.

3) Test objects: if a new object is needed as a parameter while amplifying method calls, DSpot creates an object of the correct type. In the same way, when a new method call needs primitive value parameters, DSpot generates a random value.

For example, if an *I-Amplification* is applied on the example presented in Listing 3.2, it may generate a new method call on *tl*. In Listing 3.3, the added method call is "removeAll". During this process, DSpot removes existing assertions since they might fail because it changes the state of the program.

Listing 3.3: An example of an *I-Amplification*: the amplification added a method call to *removeAll()* on *tl*.

```
testIterationOrder() {
TreeList tl=new TreeList();
tl.add(1);
tl.add(2);
tl.removeAll(); // method call added

// removed assertions
}
```

At each iteration, DSpot applies all kinds of *I-Amplification*, resulting in a set of input-amplified test methods. From one iteration to another, DSpot reuses the previously amplified tests, and further applies *I-Amplification*. By doing this, DSpot explore more the input space. The more iteration DSpot does, the more it explores, the more it takes time to complete.

3.2.2 Assertion Improvement Algorithm

***A-Amplification*:** for Assertion Amplification, is the process of automatically creating new assertions. In DSpot, assertions are added on objects from the original test case, as follows:

1) it instruments the test methods to collect the state of a program after execution (but before the assertions), i.e. it creates observation points. The state is defined by all values returned by getter methods.

2) it runs the instrumented test to collect the values. This execution results in a map per test method, that gives the values from all getters.

3) it generates new assertions in place of the observation points, using the collected values as oracle.

In addition, when a new test input sets the program in a state that throws an exception, DSpot produces a test asserting that the program throws a specific exception.

Listing 3.4: In *A-Amplification*, the second step is to instrument and run the test to collect runtime values.

```
testIterationOrder() {
TreeList tl=new TreeList();
tl.add(1);
tl.add(2);aampl
tl.removeAll();

Observations.observe(tl.size());  // logging current behavior
Observations.observe(tl.isEmpty());
}
```

For example, let consider *A-Amplification* on the test method of the example above. First, in Listing 3.4 DSpot instruments the test method to collect values, by adding method calls to the objects involved in the test case. Second, the test with the added observation points is executed, and subsequently, DSpot generates new assertions based on the collected values. In Listing 3.5, DSpot has generated two new assertions.

Listing 3.5: In *A-Amplification*, the last step is to generate the assertions based on the collected values.

```
testIterationOrder() {
TreeList tl=new TreeList();
tl.add(1);
tl.add(2);
```

```
5  tl.removeAll();
6
7  // generated assertions
8  assertEquals(0, tl.size()); // generated assertions
9  assertTrue(tl.isEmpty()); // generated assertions
10 }
```

3.2.3 Pseudo-algorithm

Algorithm 1 Main amplification loop of DSpot.

Require: Program P
Require: Test suite TS
Require: Test criterion TC
Require: Input-amplifiers $amps$ to generate new test data input
Require: n number of iterations of DSpot's main loop
Ensure: An amplified test suite ATS

1: $ATS \leftarrow \emptyset$
2: **for** t in TS **do**
3: $U \leftarrow generateAssertions\,(t)$
4: $ATS \leftarrow \{x \in U | \text{x improves TC}\}$
5: $TMP \leftarrow ATS$
6: **for** $i = 0$ to n **do**
7: $V \leftarrow []$
8: **for** amp in $amps$ **do**
9: $V \leftarrow V \cup amp.apply\,(TMP)$
10: **end for**
11: $V \leftarrow generateAssertions\,(V)$
12: $ATS \leftarrow ATS \cup \{x \in V | \text{x improves TC}\}$
13: $TMP \leftarrow V$
14: **end for**
15: **end forreturn** ATS

Algorithm 1 shows the main loop of DSpot. DSpot takes as input a program P, its test suite TS and a test-criterion TC. DSpot also uses an integer n that defines the number of iterations and a set of input-amplifiers amp. DSpot produces an amplified test suite ATS, i.e. a better version of the input test suite TS according to the specified test criterion TC. First, DSpot initializes an empty set of amplified test methods ATS that will be outputted (Line 1). For each test case t in the test suite TS (Line 2), DSpot first tries to add assertions without generating any new test input (Line 3), method $generateAssertions\,(t)$ is explained in subsection 3.2.2. It adds to ATS the tests that improve the test-criterion(Line 4).

Note that adding missing assertions is the elementary way to improve existing tests. Consequently, in DSpot there are two modes, depending on the configuration:

1) DSpot executes only assertion amplification, if $n = 0$ or $amp = \emptyset$:

2) DSpot executes both input space exploration and assertion amplification, if $n > 0$ and $amp \neq \emptyset$

In the former mode, there is no exploration of the input space, resulting in a quick execution but less potential to improve the test-criterion. In the latter mode, the exploration, depending on n, takes times but have more potential to improve the test-criterion.

DSpot initializes a temporary list of tests TMP with elements from ATS, if any (Line 5). Then it applies n times the following steps (Line 6):

1) it applies each amplifier amp on each tests of TMP to build V (Line 8-9 see subsection 3.2.1 i.e. *I-Amplification*);
2) it generates assertions on generated tests in V (Line 11 see subsection 3.2.2, i.e. *A-Amplification*);
3) it keeps the tests that improve the test-criterion(Line 12).
4) it assigns V to TMP for the next iteration. This is done because even if some amplified test methods in V have not been selected, it can contain amplified test methods that will eventually be better in subsequent iterations.

3.3 User Guide

3.3.1 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).

3.3.2 Releases

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

3.3.3 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%
```

3.3.4 Command Line Usage

You can then execute DSpot by using:

```
java -jar /path/to/dspot-LATEST-jar-with-dependencies.jar --absolute-path-to-
    project-root <path>
```

Amplify a specific test class

```
java -jar /path/to/dspot-*-jar-with-dependencies.jar eu.stamp_project.Main --
    absolute-path-to-project-root <path> --test my.package.TestClass
```

Amplify specific test classes according to a regex

```
java -jar /path/to/dspot-LATEST-jar-with-dependencies.jar --absolute-path-to-
    project-root <path> --test my.package.*
java -jar /path/to/dspot-LATEST-jar-with-dependencies.jar --absolute-path-to-
    project-root <path> --test my.package.Example*
```

Amplify a specific test method from a specific test class

```
java -jar /path/to/dspot-LATEST-jar-with-dependencies.jar --absolute-path-to-
    project-root <path> --test my.package.TestClass --cases testMethod
```

3.3.5 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.

3.3.6 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=
    JacocoCoverageSelector
```

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

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

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 option 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.package.MyTestClass2*
- *–output-path output* gives *-Doutput-path=output*

3.4 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 kinds 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%.

3.4.1 Modules descriptions

Figure 3.3 exposes the modules of DSpot. The core module is named *DSpot* which the module that amplifies existing test suites. *DSpot-diff-test-selection* is a module that selects the test methods according to a diff. This is done in order to provide a small and specific set of the existing test methods to DSpot. These test methods are executing lines that have been changed in a commit. Then, there is *DSpot-prettifier* which a module that aims at prettifying the output test methods by DSpot. Prettifying means here removing redundants statements, removing assertions that are not contributing to the quality of the amplified test method regarding the test criterion, and renaming variables and amplified test methods. This module is still under developpement and is a research prototype. Then, there is the *DSpot-maven* module that provides a maven plugin to use DSpot on maven projects. This modules aims at simplying the usage of DSpot on the command line. The *DSpot-web* provides a web interface to use DSpot remotely. Thus module communcates the result with emails. Eventually, the *Kubernetes-support* provides a parallelization of DSpot's execution.

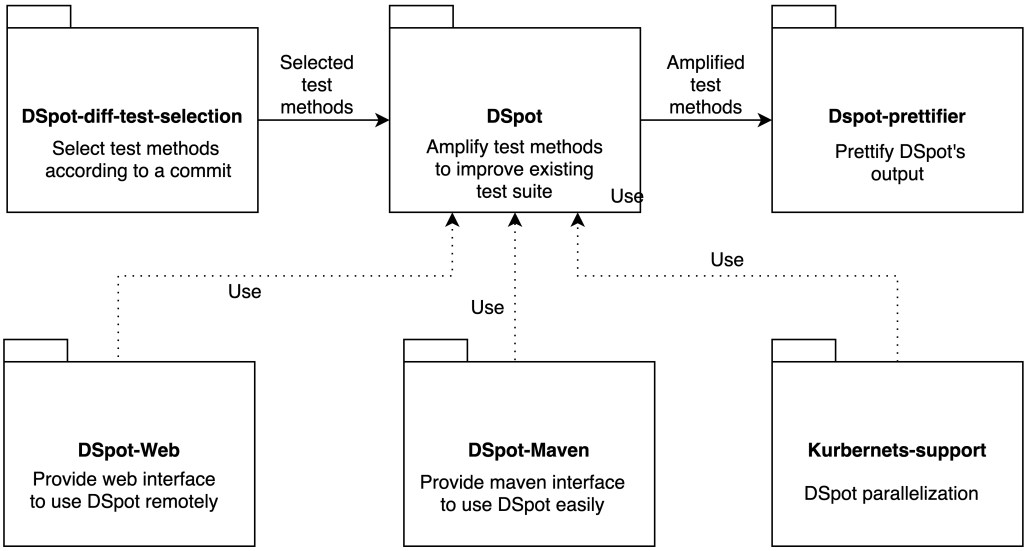

Figure 3.3: Global architectures of DSpot's modules

Table 3.2: Development statistics

Number of tests	190
Number of commits	682
Number of branches	2
Number of contributors	18
Number of pull requests	566
Number of releases (Github)	19
Number of releases (Maven Central)	12
Number of LoC (Java)	11524

3.4.2 Global workflow

Figure 3.4 exposes the normal workflow of DSpot. First, DSpot builds and prepares the project that will be amplified. Second, it modifies the input by applying pre-defined transformations in order to trigger a new behavior. Third, it generates new assertions based on the observable states of the program, using getters for example. Eventually, it selects the amplified test methods to be kept according to a pre-defined test-criterion, such as coverage.

3.5 Development Stats

Table 3.2 presents some statistics from DSpot's project and development. These statistics have been retrieve the November 19, 2019.

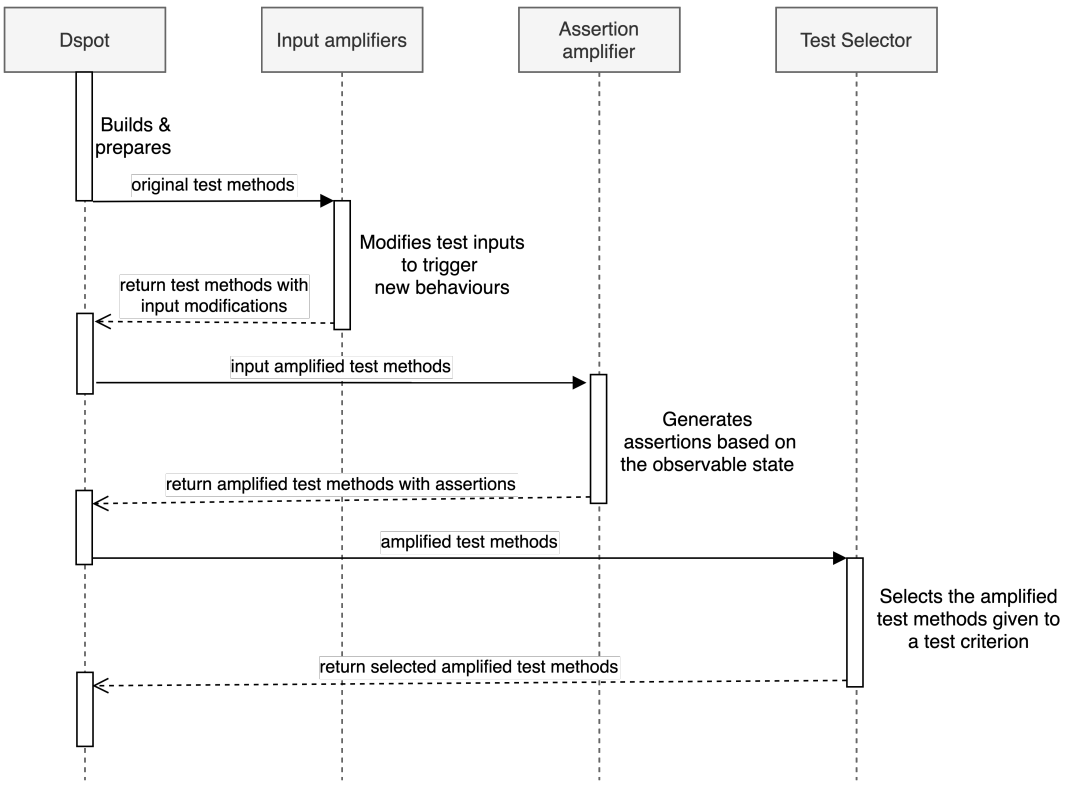

Figure 3.4: Sequence diagram of DSpot's workflow

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. These guidelines have refined over the course of the project in order to facilitate the interactions with the project partners as well as with external contributors. This constant discussion, feedback from partners and users has been instrumental to consolidate the tools and to develop relevant features.

4.1 Publications for WP1, Oct-Nov 2019

Accepted for publication in November 2019

B. Danglot, M. Monperrus, W. Rudametkin, and B. Baudry. An Approach and Benchmark to Detect Behavioral Changesof Commits in Continuous Integration. *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.

[4] P. Tonella. Evolutionary testing of classes. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 119–128, New York, NY, USA, 2004. ACM.

[5] T. Xie. Augmenting Automatically Generated Unit-test Suites with Regression Oracle Checking. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 380–403, 2006.