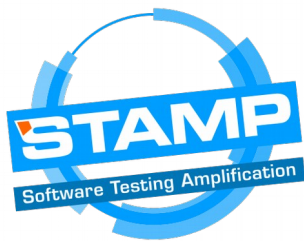


Title:	WP4 – Deliverable 4.2 – First public version of API and initial implementation of services and course-ware
Date:	February 8, 2018
Writer:	Daniele Gagliardi (ENG)
Authors:	Daniele Gagliardi (ENG), Nicola Bertazzo (ENG), Benjamin Danglot (INRIA), Luca Andreatta (ENG), Stéphane Lauriere (OW2), Olivier Bouzereau (OW2)
Reviewers:	Benoit Baudry (KTH), Caroline Landry (INRIA), Jesus Gorrongoitia (ATOS)

Table Of Content

1. EXECUTIVE SUMMARY.....	2
2. REVISION HISTORY.....	2
3. OBJECTIVES.....	3
4. INTRODUCTION.....	3
5. REFERENCES.....	3
6. ACRONYMS.....	4
7. AMPLIFICATION SERVICES.....	4
7.1. Public APIs.....	4
7.1.1. Unit test amplification.....	7
7.1.1.1. STEP 1: UPLOAD (POST) TEST CASES TO AMPLIFY.....	7
7.1.1.2. STEP 2: DOWNLOAD (GET) AMPLIFIED TEST CASES.....	7
7.1.2. Test configuration amplification.....	7
7.1.2.1. STEP 1: UPLOAD (POST) TEST CONFIGURATIONS TO AMPLIFY.....	8
7.1.2.2. STEP 2: DOWNLOAD (GET) AMPLIFIED TEST CONFIGURATIONS.....	8
7.1.3. Online amplification.....	8
7.1.3.1. STEP 1: UPLOAD (POST) CRASH INFORMATION FOR AUTOMATIC CRASH REPRODUCTION.....	8
7.1.3.2. STEP 2: DOWNLOAD (GET) GENERATED TEST CASES ABLE TO REPRODUCE THE CRASH.....	8
7.2. Maven plug-in.....	8
7.2.1. Example usage.....	11
7.3. STAMP services packaging as Docker images.....	11
8. COURSE-WARE.....	13
8.1. Tutorials.....	13
8.1.1. DSpot.....	13
8.1.2. Descartes.....	13
8.2. STAMP Virtual Lab.....	13
9. CONCLUSIONS.....	20
10. APPENDICES.....	22
10.1. DSpot Maven plug-in parameters and default values.....	22



1. Executive Summary

This report presents the first release of STAMP components developed to integrate STAMP amplification services within developers' and DevOps tool-chains, along with documentation and course-ware available to understand their usage and apply them within software life-cycle processes.

After a short introduction that describes STAMP tools developed until the end of December (tools offering unit test amplification functionality, tools to amplify test configurations and execute test cases against them, and tools that automate crash reproduction starting from production logs), the reader will be presented with an overview of STAMP micro-service reference architecture: it will start taking account of the architecture presented in the Proposal, and then it will be showed how that vision evolved along with first results obtained during the first year and according to the features of developed STAMP tools.

First version of public APIs that will be exposed by STAMP micro-services is described, along with a reference to SwaggerHub website, where the reader could find not just the APIs themselves but also examples of calls with real data that can be used to build, for instance, micro-services client.

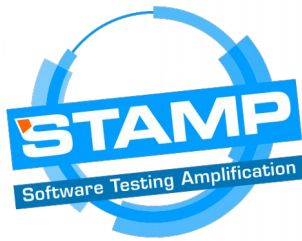
While micro-services let end users to use remotely STAMP features (in order to save computational resources in their environment), there is also the need for plug-ins that let to offer these features locally, would it be a developer workstation or a continuous integration server. For this reason a Maven plug-in was developed to expose unit test amplification features by a well-known tool, widely adopted within Java development. A Maven plug-in lets a developer use STAMP features within his workstation, as well as within a CI/CD pipeline. This plug-in now exposes DSpot amplification features with the new goal `dspot:amplify-unit-test`. This development was possible thanks to a collaboration with INRIA who made a refactoring to DSpot project, in order to make it available as a Maven dependency (a Maven repository was set up by INRIA within STAMP Github public repository). Other Maven plug-ins are planned to be developed during this year to expose also all other STAMP tools. Gradle plug-ins are also in the plan, and this should also ease the integration of STAMP tools within IDE (such as Eclipse, Netbeans, IntelliJIdea).

In addition to these developments, also several tutorials were written and published about some STAMP tools: at the moment an interested user can find within the Github repository some "for dummies" tutorial to familiarize himself with unit test amplification. In this report is briefly described the content of these tutorial and a reference were to find them.

Finally a virtual machine was prepared, named STAMP Virtual Lab, containing STAMP artifacts (tools and the Maven plug-in), in order to lets interested people to start immediately with a ready for use STAMP environment: with the collaboration of OW2 partner several USB keys containing the STAMP Virtual Lab were made available in a public session with external interested people during the STAMP in-person meeting held in Madrid in December 2017.

2. Revision History

Date	Version	Author	Comments
12-jan-2018	0.1	Daniele Gagliardi (ENG), Nicola Bertazzo (ENG), Luca Andreatta (ENG)	First draft.
24-jan-2018	0.2	Daniele Gagliardi (ENG)	Reviewed version after first internal revision
26-jan-2018	0.3	Daniele Gagliardi (ENG), Valentina Di Giacomo (ENG)	New version after second internal revision



31-jan-2018	0.4	Daniele Gagliardi (ENG)	Added reference to STAMP usb keys containing STAMP Virtual Lab and the Maven repository setup by INRIA to expose DSpot as a Maven dependency, needed for Maven DSpot plug-in
1-feb-2018	0.5	Daniele Gagliardi (ENG)	Detailed description of initial STAMP micro-services reference architecture. Refined future development section about public APIs
8-feb-2018	1.0	Daniele Gagliardi (ENG)	Final release with minor fixes and a more detailed description about future works.

3. Objectives

The objective of this report is to provide STAMP end users (developers, technical leaders, project managers) with a reference on how to integrate STAMP unit test amplification tools in their tool-chains, in order to leverage test amplification services to enhance software quality. STAMP tools can be potentially used in several ways, as standalone command line tools, by the means of Maven and Gradle plug-in, within an IDE with ad hoc extensions. A remote execution is also possible thanks to several micro-services with a well defined contract: remote usage permits to reserve local computational resources for other development tasks, assigning test amplification tasks (that can be computationally intensive) to dedicated servers. With the overview and the state-of-the-art provided here, STAMP end users should have at the end a clear vision about how to integrate STAMP tools in their workstations or in CI/CD pipelines.

4. Introduction

The crux of the WP4 ("Integration") of STAMP project is to provide individual developers and continuous integration/delivery pipelines with functionalities able to automatically amplify and increase existing test assets in order to enhance test coverage, using unit test amplification and detect regression bugs before production and drive down the cost of software testing.

This document provides a description of the first version of STAMP public API, using OpenAPI (<https://www.openapis.org/>) description format, available on SwaggerHub public site, in order to ease the development of test amplification micro-services that will be developed in the following months. These API are also intended to be a contract for developers who want to integrate STAMP services within their own tools.

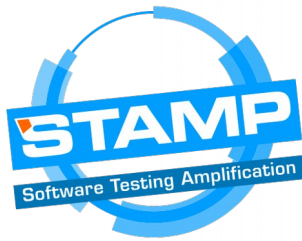
Moreover the unit test amplification Maven plug-in will be presented, to show how to use it in day-by-day activities by software developers.

STAMP artifacts require a little bit of configuration on a typical developer workstation: we prepared a virtual machine with all test amplification services and utilities ready-to-use to let developers familiarize themselves with them with no need to care about configuration issues: a description of this VM ("STAMP Virtual Lab") will be provided.

Finally a short presentation of several "for dummies" tutorials will be presented that could be used for early experiments leveraging the proposed "STAMP Virtual Lab".

5. References

- [1] Project Home: <https://www.stamp-project.eu>
- [2] Project Proposal: [stamp_ec_Proposal-SEP-210342849.pdf](#)
- [3] [D41 – Collaborative Software Engineering Platform](#)
- [4] [D42 - First public version of API and initial implementation of services and course-ware](#): A link to the most recent version of this document.
- [5] [D12 – Initial prototype of the unit test amplification tool](#)



6. Acronyms

EC	European Commission
DoW	Description of Work
CI	Continuous Integration
CD	Continuous Delivery
API	Application Programming Interface
QA	Quality Assurance
WP	Work Package
VM	Virtual Machine

7. Amplification Services

At the end of STAMP project first year, several core amplification components were designed and developed within work packages WP1, WP2 and WP3:

- unit test amplification components:
 - DSpot: it's a tool that takes as input a Java project and its test suite and produces test improvements, providing new inputs (that lead to new execution paths) and new assertions, using test mutations to generate new tests;
 - Descartes: it's a tool that, applying extreme mutation (extreme mutation removes all instructions in a method and, if the method is non-void, substitutes return value with a predefined value), let to assess the robustness of existing test cases (if the test case doesn't fail after the extreme mutation, it means that that test case itself showed its weakness being not able to detect the mutation). Using Descartes with DSpot leads to have a wider set of new tests;
- test configuration components:
 - CAMP (Configuration AMPLification): it's a tool able to generate new Docker files starting from existing ones. If Docker is used to define test environments, this tool lets you to have more test configurations in order to test your application in different environments;
 - TECOR (Test Configuration executoR): this tool lets you to execute test cases against multiple configurations. Used with CAMP, lets you to leverage amplified test configurations automating the execution process;
- runtime amplification services:
 - Evocrash: this tool is able to generate test cases starting from a Java project and a log containing a stack-trace error. If you have a stack-trace error in your production log files, it means that a bug wasn't detected by your test cases before production: this tool automate the process of producing test cases from production errors, in order to reproduce them and debug the affected source.

All these tools can be executed locally as CLI (Command-line Interface) executable.

7.1. Public APIs

According to the second methodological pillar[2], described in , STAMP services will be delivered at the end of the project, within a micro-service architecture, in which all STAMP assets are loosely coupled.

Micro-services let to integrate STAMP features in a SAAS (Software-as-a-Service) fashion: in this way computational resources usage can be optimized, leaving amplification tasks to dedicated machines. Another benefit is that a STAMP user doesn't need to configure anything on his environment: this scenario is particularly important in software companies, where software developers would leverage STAMP amplification services in order to extend their test cases and test configurations. Having these services available as SAAS, promotes their adoption.

Each of the three amplification services would be packaged as an independent, stand-alone service, so it would be possible to arrange CI/CD pipelines with desired amplification features: starting from unit test amplification, and possibly leveraging extreme mutation testing to assess the robustness of existing test cases, it would be possible to test software against different, amplified test configurations. Finally, with the availability of production logs, it would be possible to automate crash reproduction, in order to ease the investigation of production bugs, and then increase the test cases regression pool. Each of the three amplification services should be made of several "core" micro-services, specialized in a well-defined task. Proper orchestration of these micro-services leads to have more complex scenarios than the ones described here, leading to a very flexible solution. This vision, described in details in [2], is depicted in Figure 1:

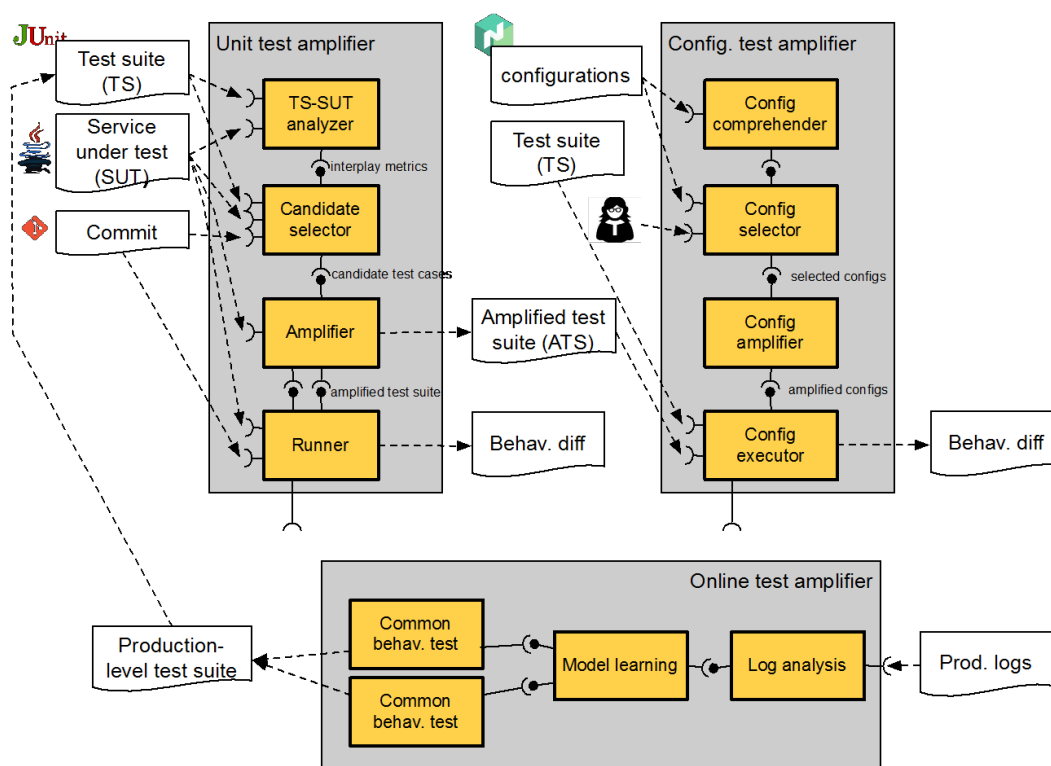


Figure 1: STAMP initial micro-service architecture (source: STAMP EC Proposal [2])

Each amplification service is decomposed in a flow of micro-services:

1. in the unit test amplifier, a micro-service analyses the application under test (SUT Analyzer). Another micro-service (Candidate Selector) makes a selection among its test cases to candidate them for the amplification. At the end two other micro-services respectively amplify (Amplifier) selected test cases and execute them (Runner);
2. Configuration test amplifier leverages several micro-services that analyse available configurations, for instance in form of Docker images (Config comprehender), select configurations to candidate

for amplification (Config Selector), amplify them (Config Amplifier) and then execute test cases against them (Config Executor);

- Online test amplifier is made of micro-services which analyse production logs containing crash stack-traces (Log Analyzer), analyse related source code to understand how to reproduce the crash (Model Learner) and generate test cases able to reproduce the error condition that led to the crash (Behaviour Tester).

Core amplification components described in the previous section now offer amplification capabilities needed to develop these micro-services (some of these tools are quite mature and have well defined contracts to be used as integration point). Engineering team was involved in several development activities (mainly in DSpot and Descartes development) during the last year, and this experience provided Engineering team with a more clear insight about amplification “internals” and which integration points are available to implement required micro-services.

During the development of core amplification components, the definition of public APIs started in parallel, in order to have an early version of the contract micro-services should expose.

At the moment three services have been defined to expose amplification features. The following figure depicts this concept, showing these three new services (Unit test amplifier, Test configuration amplifier, Online amplifier) and their role within the general STAMP scenario:

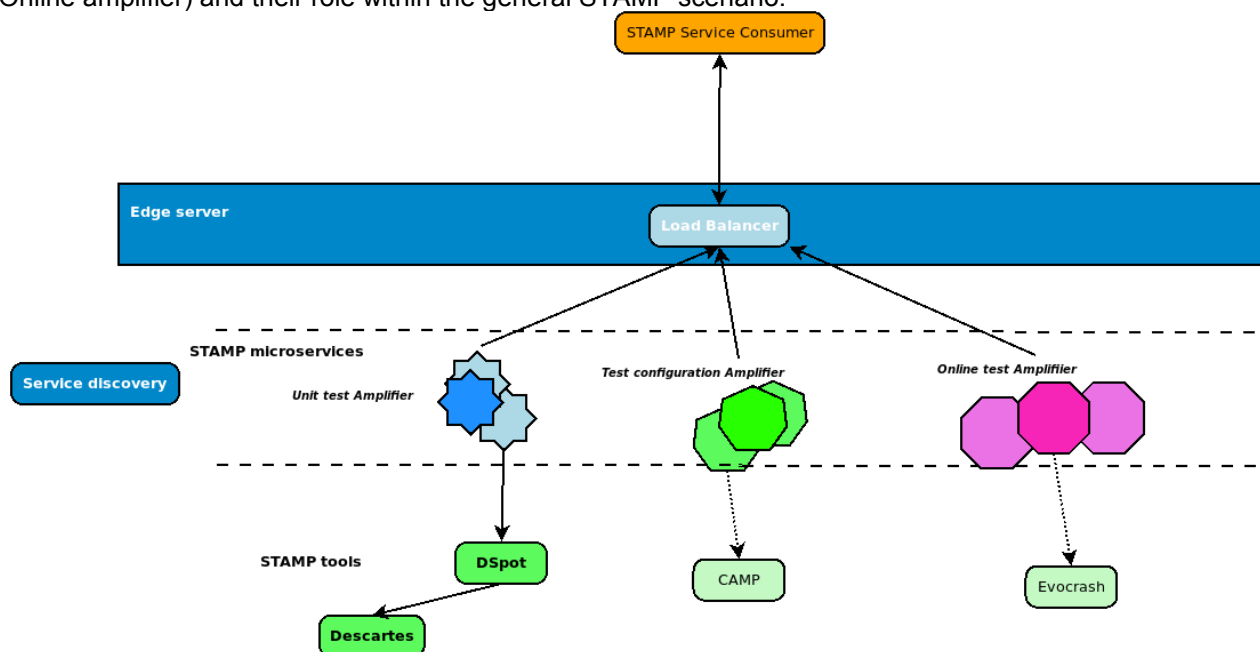
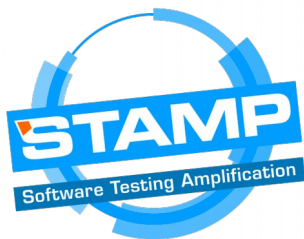


Figure 2: STAMP tools exposed by wrapper micro-services

At the moment there isn't a direct usage of Descartes extreme mutation testing features, but it can be used as one among mutators available to DSpot.

Following a “Design first” approach, API have been defined by the means of SwaggerHub API design functionalities (<https://app.swaggerhub.com/apis/stamp-project/>): each amplification service is accomplished with a two-steps action:

- upload test cases/test configurations/production logs as input to STAMP tools;
- get amplified test cases/test configurations.



In this release unit test amplification (provided by DSpot) is explicitly supported by APIs, while other tools will be supported in next releases (plans are by the end of 2018).

Moreover, in the current API release all tool parameters aren't explicitly passed to micro-services; default values are used instead.

7.1.1. Unit test amplification

Unit test amplification service has the `/javaprojects` endpoint. It inherits the contract from DSpot and exposes two methods:

- POST: upload the Maven project, containing test cases to amplify;
- GET: download amplified test cases.

In this release the upload of the project working copy is supported, while in the next release it will be possible to specify the URL of SCM repository hosting the source code. The two approaches have pros and cons:

- amplify test cases from a working copy:
 - pros: the developer can amplify test cases from her working copy and then decide whether push them in the remote repository or not;
 - cons: lot of data need to be uploaded from developer working station to the remote amplification service;
- amplify test cases from a SCM repository:
 - pros: few data need to be sent to the remote amplification service;
 - cons: test cases from a working copy cannot be amplified so the developer should push her working copy before asking for test amplification.

7.1.1.1. STEP 1: upload (POST) test cases to amplify

POST method accepts a compressed file, in Base 64 format, containing the source code, as input parameter. Two possible responses are available:

- success: project uploaded (code 201). The response contains the identifier assigned to the uploaded project (it's a MD5 hash computed over the input parameter): `{"id" : "E85C24916AB75382FE1BC9235406B400"}`;
- failure: project already exists (code 409). In this case a response in JSON format is returned: `{"message" : "Duplicate Java project found" }`

In next releases other error conditions will be supported, with proper codes and messages.

7.1.1.2. STEP 2: download (GET) amplified test cases

GET methods accepts the uploaded project id and returns a compressed file in Base 64 format, containing amplified test cases and the report generated by DSpot. Two possible responses are available:

- success: code 200. The response in JSON format contains the DSpot output directory, compressed and encoded in Base 64 format;
- failure: code 404. The required project doesn't exist. The response in JSON format contains a proper message `{"message" : "Project not found" }`.

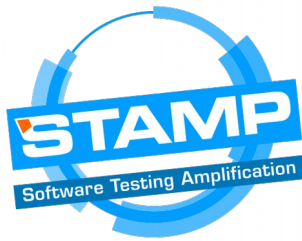
In next releases other error conditions will be supported, with proper codes and messages.

Implementation will require the development of two distinct micro-services, one to upload test cases to amplify, and another one to download amplified test cases.

7.1.2. Test configuration amplification

Test configuration amplification service has the `/testconfigurations` endpoint. It exposes two methods:

- POST: upload test configurations to amplify. Current version supports test configurations made of, made of one or two Docker files (a Docker file and optionally a Docker Swarm file);
- GET: download amplified test configurations.



7.1.2.1. STEP 1: upload (POST) test configurations to amplify

POST method accepts a compressed file, in Base 64 format, containing Docker files, as input parameter. Two possible responses are available:

- success: test configuration uploaded (code 201). The response contains the identifier assigned to the uploaded test configuration (it's a MD5 hash computed over the input parameter): {"id" : "aff788bbfdfaeada4c33ef2460bbcf56"};
- failure: test configuration already exists (code 409). In this case a response in JSON format is returned: {"message" : "Duplicate test configuration found" }

In next releases other error conditions will be supported, with proper codes and messages.

7.1.2.2. STEP 2: download (GET) amplified test configurations

GET methods accepts the uploaded project id and returns a compressed file in Base 64 format, containing amplified test cases and the report generated by DSpot. Two possible responses are available:

- success: code 200. The response in JSON format contains the amplified unit tests;
- failure: code 404. The required project doesn't exist. The response in JSON format contains a proper message {"message" : "Project not found" }.

In next releases other error conditions will be supported, with proper codes and messages.

Implementation will require the development of two distinct micro-services, one to upload test configurations, and another one to download amplified test configurations.

7.1.3. Online amplification

Unit test amplification service has the `/onlineamplifications` endpoint. It exposes two methods:

- POST: upload a crash stack-trace and the source code which caused the crash;
- GET: download newly generated test cases.

7.1.3.1. STEP 1: upload (POST) crash information for automatic crash reproduction

POST method accepts a compressed file, in Base 64 format, containing the log excerpt containing crash information and the affected source code, as input parameter. Two possible responses are available:

- success: crash information uploaded (code 201). The response contains the identifier assigned to crash information (it's a MD5 hash computed over the input parameter): {"id" : "F5509EBEB022A12FD0A4233CB035F56D"};
- failure: duplicate crash information found (code 409). In this case a response in JSON format is returned: {"message" : "Duplicate crash information found" }

In next releases other error conditions will be supported, with proper codes and messages.

7.1.3.2. STEP 2: download (GET) generated test cases able to reproduce the crash

GET methods accepts the uploaded crash id and returns a compressed file in Base 64 format, containing newly generated test cases. Two possible responses are available:

- success: code 200. The response in JSON format contains the newly generated unit tests;
- failure: code 404. The required crash doesn't exist. The response in JSON format contains a proper message {"message" : "Crash information not found" }.

In next releases other error conditions will be supported, with proper codes and messages.

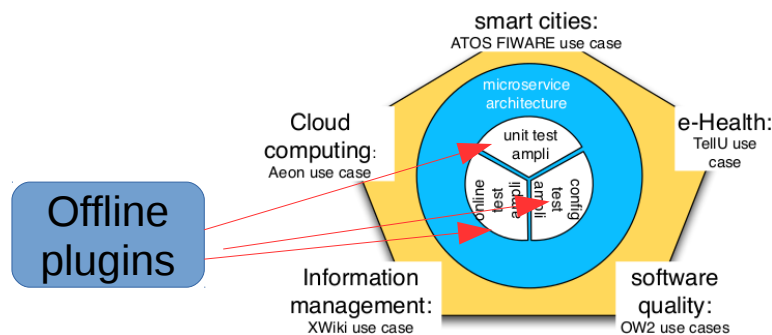
Implementation will require the development of two distinct micro-services, one to upload crash information, and another one to download generated test cases.

7.2. Maven plug-in

In the first half of 2017 several interviews were conducted with STAMP use case owners, and it emerged that having the possibility to use STAMP tools also in off-line mode would increase tools flexibility:

- to overcome network issues (firewalls, proxies, connection issues), it would be better to use test amplification services in a off-line fashion;
- amplification tasks could affect development environment performances, so using STAMP tools as SAAS can address this issue;

So Maven plug-in development started to provide off-line amplification capabilities.



The decision to start with Maven plug-in development was enforced by the consideration that it can be leveraged in several different scenarios:

- direct usage (mvn command line)
- within an IDE supporting Maven (i.e. Eclipse M2E, IntelliJIdea, Netbeans);
- within a CI/CD pipeline (usually continuous integration solutions such Jenkins support Maven goals out-of-the-box).

To some extent, a Maven plug-in exposing a single amplification service locally within a developer workstation can be considered a “micro-service”, being a fine grained component, specialized in one specific function. The very nature of Maven is modular itself, reinforcing this vision.

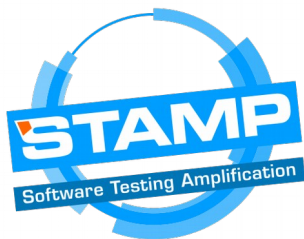
A Maven plug-in was developed to expose DSpot unit test amplification feature in a more familiar way for a developer. Source code for this plug-in is available at <https://github.com/STAMP-project/dspot/tree/master/dspot-maven>.

This plug-in exposes the “amplify-unit-tests” goal that lets a developer to run DSpot amplification. The usage is very simple: from the Maven Java project root folder (which contains the `pom.xml` file) just run the following command:

```
mvn dspot:amplify-unit-tests
```

DSpot parameters are passed as Maven parameters. To use it it’s necessary to add following information to `pom.xml` file:

```
<plug-in>
  <groupId>fr.inria.stamp</groupId>
  <artifactId>dspot-maven</artifactId>
  <version>1.0.4-SNAPSHOT</version>
  <configuration>
    <goalPrefix>dspot</goalPrefix>
  </configuration>
</plug-in>
```



```
.....  
</configuration>  
</plug-in>
```

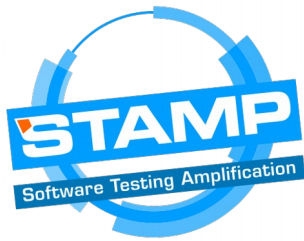
The parameter `<goalPrefix>` lets the developer execute the unit test amplification goal with a short name (otherwise he would have to use the more verbose notation `groupId:artifactId:version`).

In the current release not all DSpot parameters are passed as Maven parameters: during the 2018 the support to all DSpot parameter will be completed. Following there is a brief description of supported parameters (for a detailed description refer to [5] and Github official repository <https://github.com/STAMP-project/dspot>):

- `<amplifiers>`: this parameter lets to specify which set of amplifiers should be used. Possible values are: `NumberLiteralAmplifier`, `MethodAdd`, `MethodRemove`, `TestDataMutator`, `StatementAdd`, `None`. Default value is `MethodAdd`;
- `<iteration>`: specify the number of amplification iteration. The more iterations, the more improvement in test criterion (a larger number of iterations could kill more mutants), but of course this has an impact on the execution time. Default value is 3 (three iterations);
- `<test-cases>`: a list containing fully qualified names of test cases to amplify. Default to `all` (`all` is a placeholder meaning that all test classes need to be amplified);
- `<test-criterion>`: this parameter defines the test adequacy criterion to maximize with amplification. It takes an indicator (mutation score, number of executed mutant, test coverage, etc) and uses it to evaluate the adequacy of newly generated test cases. Possible values are `PitMutantScoreSelector`, `ExecutedMutantSelector`, `CloverCoverageSelector`, `BranchCoverageTestSelector`, `JacocoCoverageSelector`, `TakeAllSelector`, `ChangeDetectorSelector`. Default value is `PitMutantScoreSelector`;
-
- `<output-path>`: Java project sub-folder where DSpot report will be stored. Default is `dspot-report` within root project folder (the folder containing the `pom.xml` file);
- `<randomSeed>`: specify a prime number seed for randomness used within DSpot operations. Default to 23;
- `<timeOut>`: specify a timeout (in milliseconds) for test executions. Default to 10000 ms (10 seconds); A generated test can lead to a never-ending loop, so a generated test will be killed after this timeout;
- `<project>`: project root dir. Default to Maven project base directory;
- `<src>`: project folder containing sources. Default to Maven project sources directory;
- `<test>`: project folder containing test cases. Default to Maven test sources directory;
- `<classes>`: project folder containing compiled project classes. Default to `"target/classes"` within project root folder;
- `<testClasses>`: project folder containing compiled test classes. Default to `"target/classes"` within project root folder
- `<tempDir>`: temporary folder required by DSpot. Default to `"tempDir"` within project root folder;
- `<filter>`: filter on package containing test classes to amplify
- `<mavenHome>`: path to Maven Home. It's required by DSpot (invoked by within Maven plug-in)
-

Subsequent releases will add support to following parameters:

- `<max-test-amplified>`: specify the maximum number of amplified test that DSpot keeps (before generating assertion). Default value is 200 amplified tests;



- `<path-pit-result>`: specify the path to the .csv of the original result of Pit Test;
- `<useReflection>`: use a isolated test runner, leveraging reflection features of Java language;
- `<excludedClasses>`: classes to be excluded from DSpot analysis
- `<additionalClasspathElements>`: additional classes and libraries needed for build and execution;
- `<excludedTestCases>`: test cases to be excluded from amplification;

In appendix 10.1 a pom.xml fragment is reported, containing all parameters along with their default value.

The development of DSpot Maven plug-in required a refactoring on DSpot project itself, in order to use it as a Maven dependency. ENG and INRIA made a joint effort to accomplish this refactoring, and INRIA added DSpot and other dependencies to the STAMP Maven repository, already available at <https://github.com/STAMP-project/stamp-maven-repository>.

The availability of a Maven repository as well as DSpot dependencies opens new trails to STAMP tools integrations, which will be developed in the current year.

7.2.1. Example usage

Suppose you want to use DSpot amplification capabilities, using DSpot Maven plug-in within the Dhell sample project and you want to amplify the HelloAppTest test case. You will need to add to Dhell POM file the following rows:

```
<plug-in>
  <groupId>fr.inria.stamp</groupId>
  <artifactId>dspot-maven</artifactId>
  <version>1.0.4-SNAPSHOT</version>
  <configuration>

    <goalPrefix>dspot</goalPrefix>

    <amplifiers>MethodAdd</amplifier>

    <test>fr.inria.stamp.examples.dhell.HelloAppTest</test>

    <filter>fr.inria.stamp.examples.dhell.*</filter>

  </configuration>
</plug-in>
```

This configuration will amplify your test case, applying three times (parameter `<iteration>` set to default value 3 – see 10.1) the DSpot “MethodAdd” amplifier. Now you just need to give following command:

```
mvn dspot:amplify-unit-tests
```

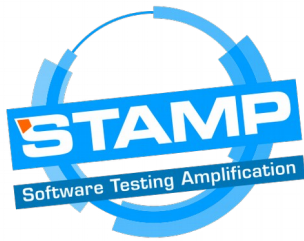
in the root folder of your project.

7.3. STAMP services packaging as Docker images

According to Task 4.2 described in [2], STAMP assets will be packaged as Docker images to ease the integration in developer and CI/CD tool-chains.

In the first half of year 2017 several experiments were carried out on Gitlab CI environment (the OW2 Gitlab instance, part of the collaborative platform described in [3]) to setup a process to package STAMP assets as Docker images.

The solution is based on the following activities:



- a Docker Gitlab Runner was installed and configured within the STAMP collaborative platform;
- `gitlab-ci.yml` file was created to configure a delivery process containing these steps:
 - **build:** executes `mvn clean compile` command into a container based on `maven:3.3.9-jdk-8` image
 - **test:** executes `mvn test` command into a container based on `maven:3.3.9-jdk-8` image
 - **package:** executes `mvn package` into a container based on `maven:3.3.9-jdk-8` image
 - **int-test:** executes `mvn verify` into a container based on `maven:3.3.9-jdk-8` image

Following there is an excerpt from the `gitlab-ci.yml` file:

`build:`

```
script: "mvn clean compile"
image: maven:3.3.9-jdk-8
tags:
  - docker
```

`test:`

```
script:
  - "mvn test"
image: maven:3.3.9-jdk-8
tags:
  - docker
```

`package:`

```
script: "mvn package"
image: maven:3.3.9-jdk-8
tags:
  - docker
```

`int-test:`

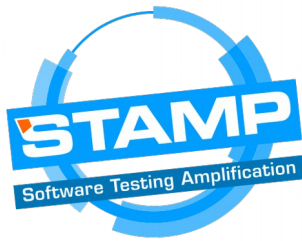
```
script: "mvn verify"
image: maven:3.3.9-jdk-8
tags:
  - docker
```

`docker-build:`

```
script:
  - "mvn package"
image: maven:3.3.9-jdk-8
tags:
- docker
```

Every section (`build`, `test`, `package`, `int-test`, `docker-build`) is a step in the CI/CD process.

The complete POC is available at <https://gitlab.ow2.org/stamp/stamp-devops-poc>



8. Course-ware

8.1. Tutorials

Several “For dummies” tutorial have been developed to show the usage of DSpot e Descartes with Maven Java projects and Gradle Java projects:

- DHELL (Maven Java project): <https://github.com/STAMP-project/dhell.git>
- DHEG (Gradle Java project): <https://github.com/STAMP-project/dheg.git>

These tutorials are publicly available on Github DSpot/docs and Descartes/docs sections.

8.1.1. DSpot

“DSpot for dummies” is available at <https://github.com/STAMP-project/dspot/blob/master/docs/dspot-for-dummies.md> and shows how to use DSpot within two kind of Java projects:

1. Maven project
2. Gradle project

In both cases the tutorial is comprised of the following steps:

1. DSpot setup: clone DSpot project and package it (`mvn package`)
2. Clone sample project
3. Execute DSpot from the sample project root

8.1.2. Descartes

Descartes uses extreme mutation testing by the means of several mutators configured before the mutation would take place.

“Descartes for dummies” show an easy way to start with extreme mutation testing concepts, using two kind of Java projects:

1. Maven project
2. Gradle project

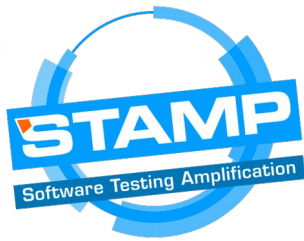
Descartes applied to a Maven Java project is available at <https://github.com/STAMP-project/pitest-descartes/blob/master/docs/Descartes-for-dummies-mvn.md> and is comprised of the following steps:

1. Descartes setup: clone Descartes and package it (`mvn package`)
2. Clone sample Maven Java project (DHELL)
3. Specify needed mutators within the target project `pom.xml` file
4. Apply Descartes to target project
5. Check the report available in the `build/reports/pitest/YYYYMMDDHHMI` project sub-folder.

Descartes applied to a Gradle Java project is available at <https://github.com/STAMP-project/pitest-descartes/blob/master/docs/Descartes-for-dummies-gradle.md> and is comprised of the following steps:

1. Descartes setup: clone Descartes and package it (`mvn package`)
2. Clone sample Gradle Java project (DHEG)
3. Specify needed mutators within the target project `build.gradle` file
4. Apply Descartes to target project
5. Chek the report available in the `build/reports/pitest/YYYYMMDDHHMI` project sub-folder.

8.2. STAMP Virtual Lab



A virtual machine equipped with all STAMP artifacts developed by December 2017 was set up and made available publicly in OW2 infrastructure: <https://nextcloud.ow2.org/index.php/s/lmbPEkUU3NoUgWm>

It requires Virtualbox, 2 GB RAM and at least 10 GB disk space. It's based on Ubuntu 16.04.3 LTS and it's equipped with:

- Oracle JDK 1.5_151
- Apache Maven 3.5.2
- Git 2.7.4
- Gradle 4.3.1

In this version, following STAMP artifacts are available to make some experience with test amplification:

- Pitest-Descartes plug-in
- DSpot (DSpot command line and DSpot Maven plug-in, described in 7.2 section)
- Dhell: a Maven toy project
- Dhegg: a Gradle toy project.

Once the virtual machine is configured, according instructions provided at , the end user can start it and access with `stamp/stamp` credentials:

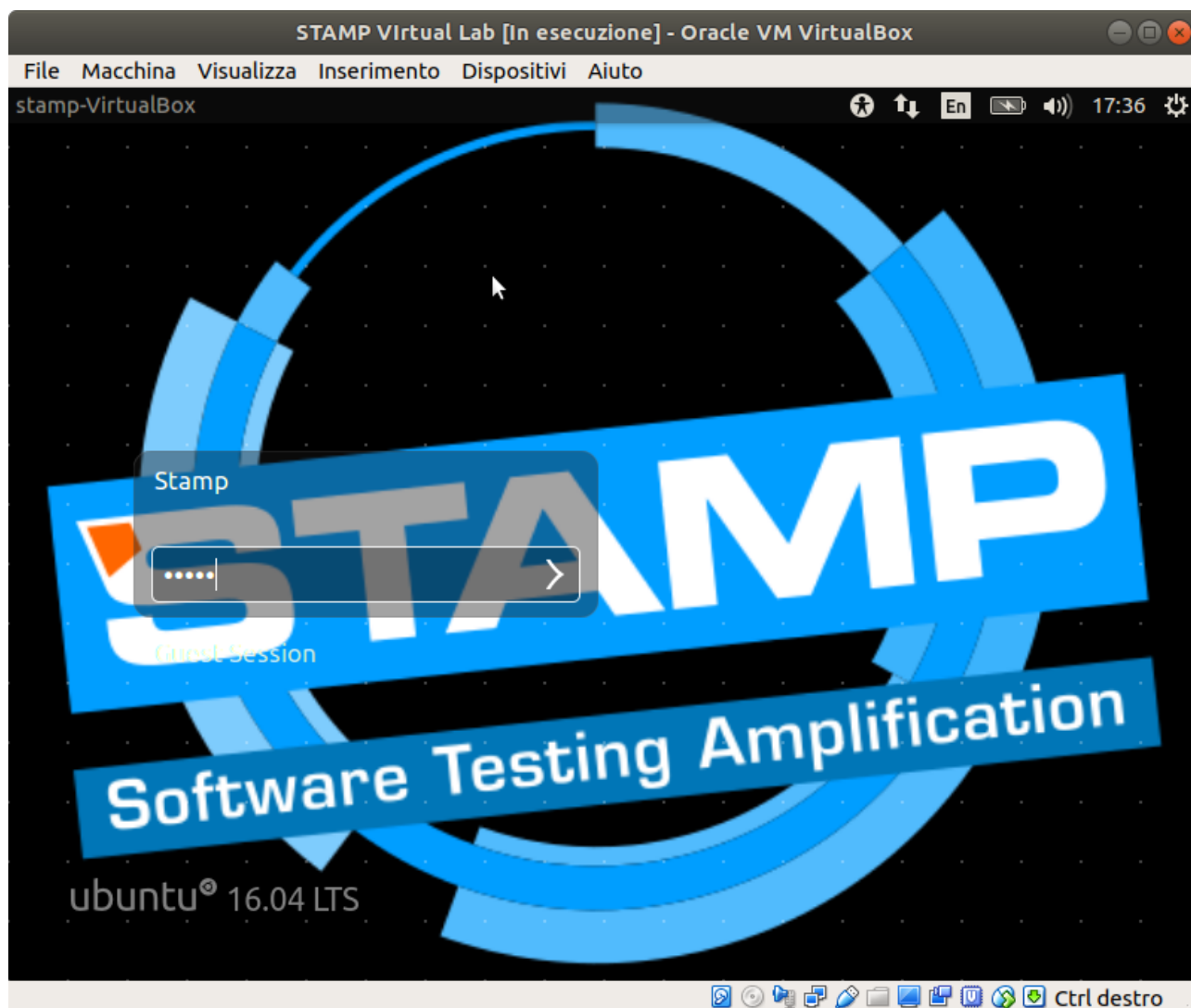
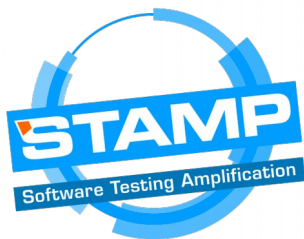


Figure 2: STAMP Virtual Lab login.

stamp user has `sudo` permissions. After login, in stamp user home a `stamp` folder is available with DSpot e Descartes installed, along with the sample projects DHELL and DHEG:

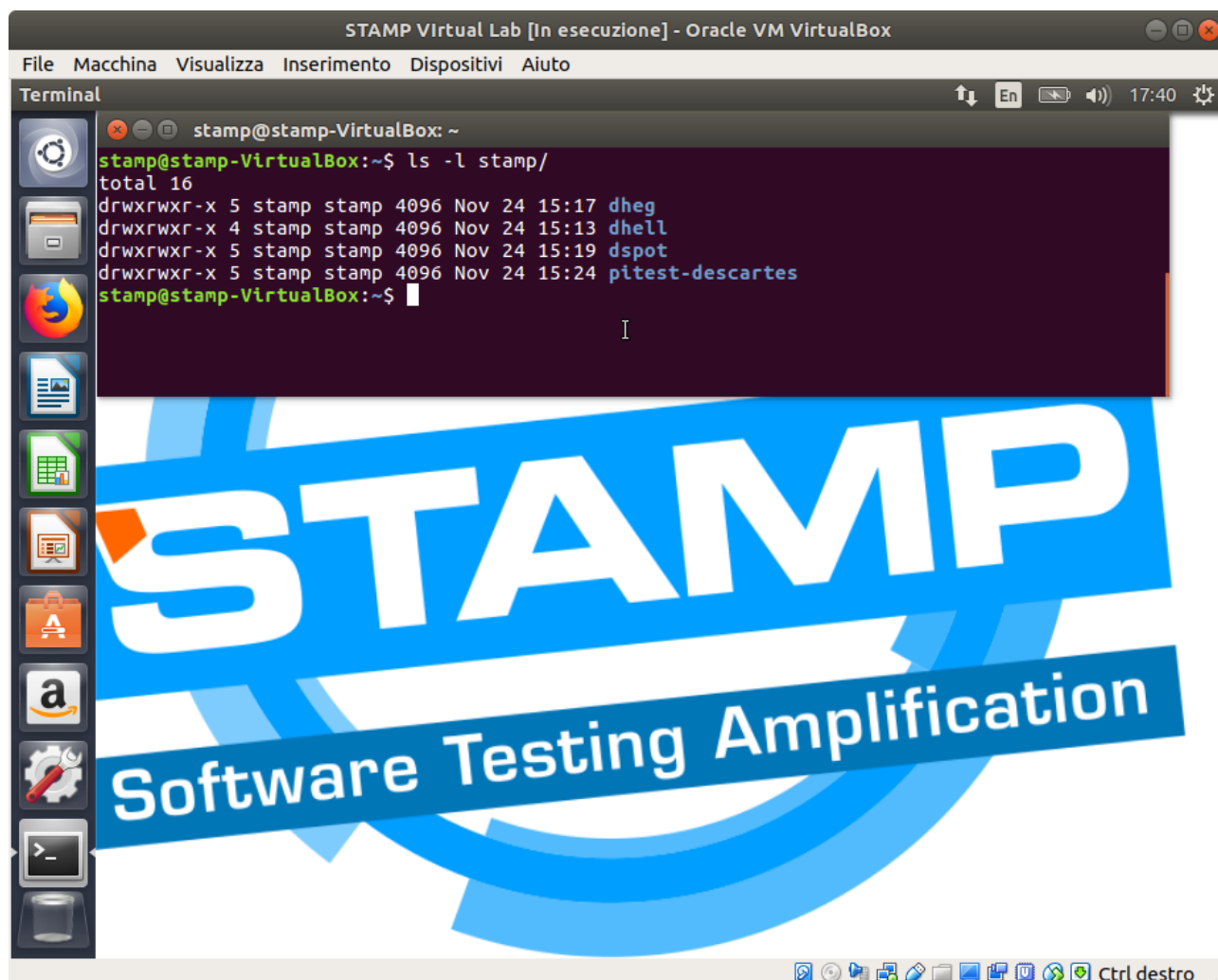


Figure 3: STAMP Virtual Lab Home

STAMP Virtual Lab contains already all the Maven dependencies to build both DSpot and Descartes, so the build process can execute even if an Internet connection isn't available.

Now it's possible to start immediately with the tutorials described in the previous section, building DSpot and Descartes, and applying them to Dhell and Dheg projects to familiarize with unit test amplification and extreme mutation testing. In a terminal go to stamp/dspot folder and give the command:

```
mvn package
```

Maven will build DSpot, and after a while you could start use it:

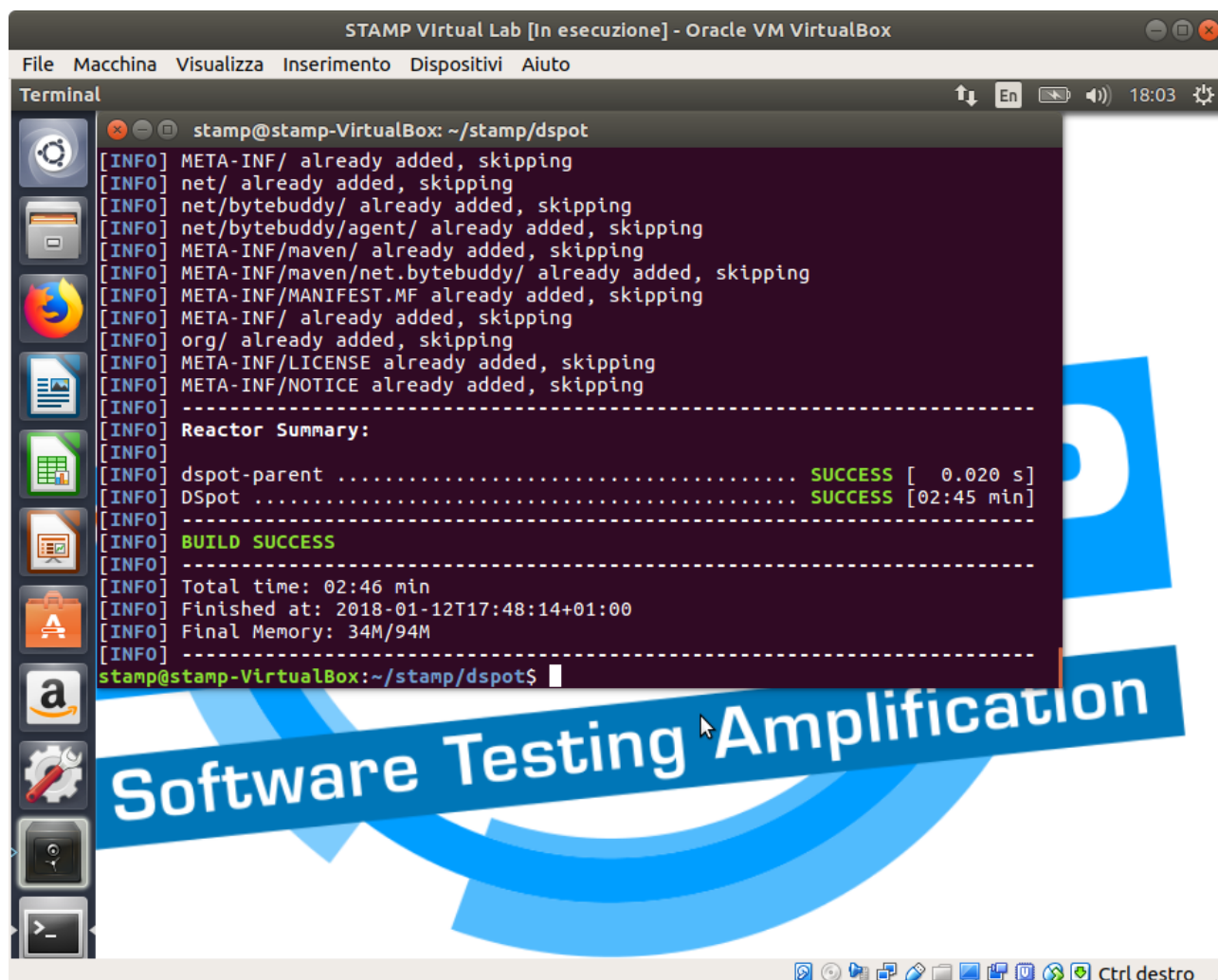


Figure 4: DSpot compilation within STAMP Virtual Lab.

On the desktop a Readme file gives short instructions to access to on-line “for dummies tutorials”. Following there is an excerpt from “DSpot for dummies” tutorial.

To start, open another terminal (or from the previous where you built DSpot) and go to `stamp/dhell` folder. Before applying DSpot, create in project root a text file containing DSpot configuration with the following content:

```

project=.
src=src/main/Java
testSrc=src/test/Java
testResources=src/test/resources
outputDirectory=target/dspot-output
filter=fr.inria.stamp.examples.*
  
```

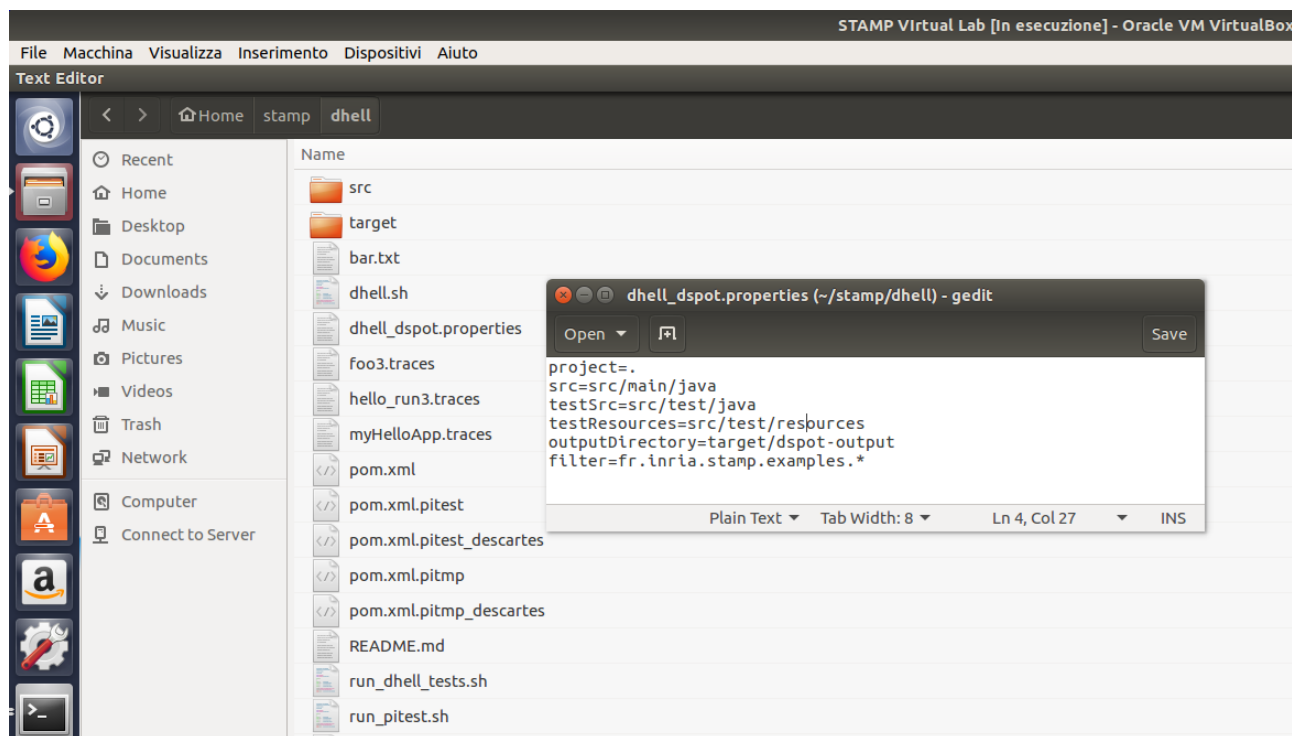
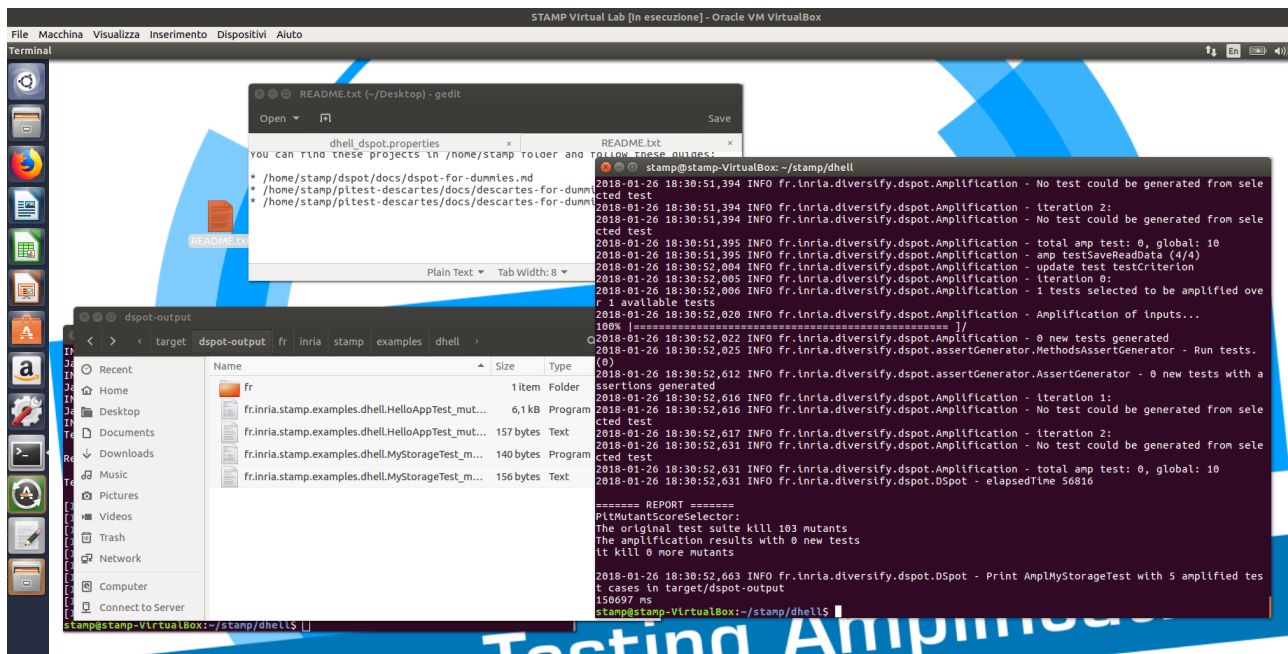


Figure 5: DSpot configuration file for DHell sample project within STAMP Virtual Lab.

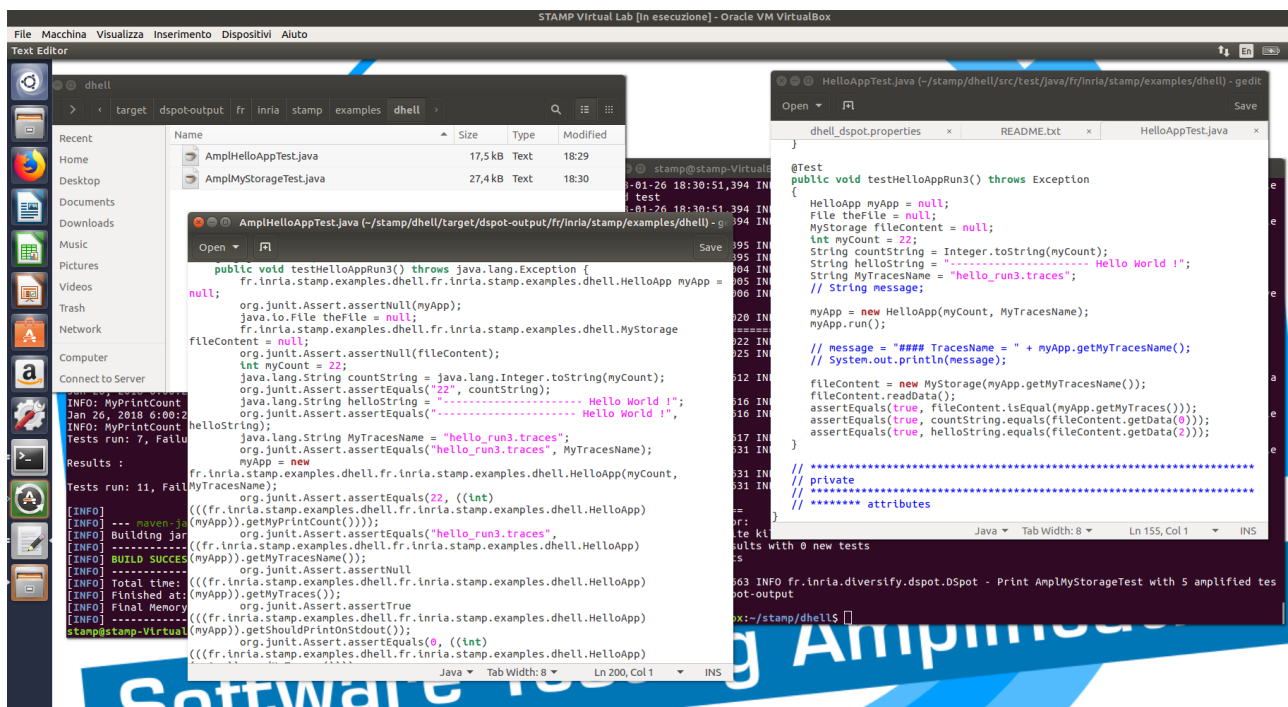
After saving this file, launch the following command:

```
Java jar ../dspot/dspot/target/dspot-1.0.0-jar-with-dependencies.jar -path-to-properties
dhell_dspot.properties
```

DSpot will start to analyze Dhell source code and test cases, it will generate new test cases and execute them, and finally it will show final results:



On the right DSpot execution is visible with amplification results. On the left the DSpot output directory content is visible along with amplification reports and a folder containing amplified test cases. During this run DSpot generated additional test cases, saving them within DSpot output directory:



On the right you can see the original test class (with a focus on testHelloAppRun3() test method, while on the left you can see the same same test method with amplification applied to it (more inputs and more assertions).

Thanks to the collaboration with OW2, this virtual machine was made available to the attendees of the public seminar, held within the Madrid in-person meeting in December 2017:



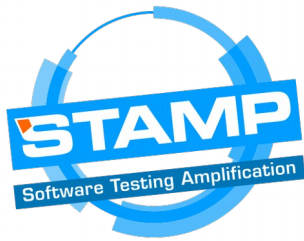
Figure 6: USB keys containing STAMP Virtual Lab VM.

9. Conclusions

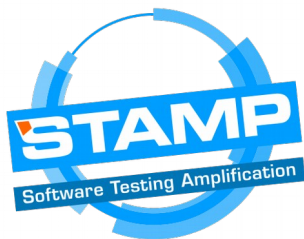
This deliverable reports on the development related to the integration of STAMP artifacts in development tool chains, and on the documentation and course-ware developed to help end users in adopting test amplification techniques.

Future work will focus on:

- definition of API for IDE client. Thanks to the collaboration with ATOS, the following activities will be accomplished:
 - Automatic launching of DSpot with default (previous) configuration (project and execution) for IDE projects managed either with Maven or Gradle.



- Wizard assisted execution configuration for DSpot. Wizard assisted project configuration for DSpot
- Management of DSpot execution profiles, and project configuration, so after selecting both, DSpot is executed with such configuration
- DSpot execution tracing in IDE console
- Consolidating DSpot execution results in IDE dedicated view. Traceability with generated tests.
- Storage and inspection of execution traces.
- Management of amplified tests within the IDE project.
- Integration of STAMP tools in different tool chains (hooks, plug-ins, etc) and CI systems (Travis, Jenkins, Gitlab CI). Next steps will include the exploitation of DSpot Maven plug-in in order to automate test amplification within Gitlab CI and Jenkins;
- Developing a platform to use STAMP tools as services (STAMP SAAS). In order to accomplish this goal, APIs development will continue with:
 - adding support to upload DSpot parameters along with source code and test cases to amplify;
 - extending unit test amplification APIs to apply DSpot amplification on test cases available in SCM repositories. This is the scenario described in 7.1.1.1;
 - adding support to Descartes extreme mutation testing, to validate existing test cases;
 - adapting current on-line amplification contract to support Evocrash features;
 - adapting current test configuration amplification contract to CAMP features.
- According to defined APIS, development of STAMP micro-services, using Seedstack (<https://seedstack.org>) framework. Seedstack is one of OW2 projects initially considered as possible STAMP use cases. It's an opinionated framework that supports best practices and standard development guidelines mainly in micro-services development. Collaboration with ActiveEon will lead to a robust micro-services implementation, thanks to ActiveEon expertise in cloud and micro-services development;
- Provide developers with easy-to-use plug-ins which will expose STAMP tools features:
 - developing a Maven plug-in to support Evocrash;
 - developing a Maven plug-in to support CAMP/TECOR;



- developing Gradle plug-in to support DSpot, Descartes, Evocrash, and CAMP/TECOR
- keeping STAMP Virtual Lab aligned with the latest versions of STAMP artifacts.

10. Appendices

10.1. DSpot Maven plug-in parameters and default values

Following an example of DSpot plug-in configuration: parameters values equal default values used by the plug-in:

```
<configuration>

<!--// Command Line parameters - fr.inria.stamp.Configuration-->

    <amplifiers>MethodAdd</amplifier>

    <iteration>3</iteration>

    <test-criterion>PitMutantScoreSelector</test-criterion>

    <test>all</test>

    <output-path>${project.build.directory}/dspace-report</output-path> <!--
project.build.directory is a Maven property-->

    <randomSeed>23</randomSeed>

    <timeOut>10000</timeOut>

    <selector>PitMutantScoreSelector</selector>

<!-- Properties file parameters - fr.inria.diversify.runner.InputConfiguration--
>

    <project>${project.basedir}</project> <!--project.basedir is a Maven
property-->

    <src>${project.build.sourceDirectory}</src><!--
project.build.sourceDirectory is a Maven property-->

    <test>${project.build.sourceDirectory}</test>

    <classes>${project.build.outputDirectory}</classes>

    <testClasses>${project.build.testOutputDirectory}</testClasses>

    <tempDir>${project.build.directory}/tempDir</tempDir>

    <filter>example.*</filter>

    <mavenHome>${env.M2_HOME}</mavenHome>

</configuration>
```