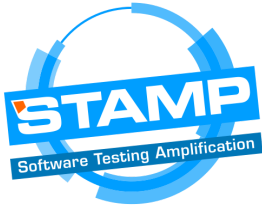




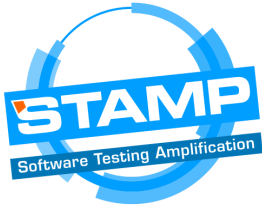
Title :	WP5 – D5.5 – Use Cases Validation Report V1
Date:	June 22, 2018
Writers:	Pascal Urso, Activeeon Mael Audren, Activeeon Jesús Gorroñoigoitia, ATOS Fernando Mendez, ATOS Lars Thomas Boye, TellU Vincent Massol, XWiki Assad Montasser, OW2 Cédric Thomas, OW2
Reviewers:	Benoit Baudry (KTH), Xavier Devroey (TUDelft), Daniele Gagliardi (ENG), Caroline Landry (INRIA)

Table Of Content

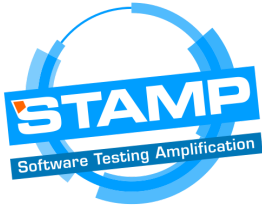
EXECUTIVE SUMMARY	4
REVISION HISTORY	4
OBJECTIVES	4
INTRODUCTION	4
REFERENCES	5
GLOBAL QUALITY MODEL	5
USE CASE : ProActive WORKFLOWS AND SCHEDULING (ActiveEon)	5
INTRODUCTION	5
EXPECTATIONS FOR STAMP TOOLS	7
VALIDATION OBJECTIVES FOR THE PERIOD	7
EvoCrash	7
CAMP	8
VALIDATION WORK	8
VALIDATION PREPARATION	8
DSpot	8
EvoCrash	8
CAMP	9
VALIDATION RESULT	9
DSpot	9
EvoCrash	9
CAMP	11
QUALITY MODEL	12



USE CASE : FIWARE ECOSYSTEM (ATOS)	13
INTRODUCTION	13
VALIDATION CONTEXT	13
ATOS OBJECTIVES FOR STAMP	14
VALIDATION WORK	15
QUALITY MODEL	15
VALIDATION OBJECTIVES	16
VALIDATION PREPARATION	16
VALIDATION RESULTS	23
DESCARTES	23
DSpot	27
CAMP	29
QUALITATIVE EVALUATION AND RECOMMENDATIONS	31
USE CASE : TELLU	33
INTRODUCTION	33
TELLUCLOUD 3.X SYSTEM	33
STAMP USE CASE FOCUS	34
TELLU OBJECTIVES FOR STAMP	35
VALIDATION OBJECTIVES FOR THE PERIOD	37
VALIDATION PREPARATION	37
CODE DEVELOPMENT	37
COLLECTING METRICS	37
TELLUCLOUD TESTING TOOL	38
COMPONENT AND SYSTEM TESTS	39
DEPLOYMENT CONFIGURATION	40
VALIDATION RESULTS	41
DESCARTES	41
DSpot	42
EvoCrash	42
QUALITY MODEL	42
USE CASE : XWiki	43
INTRODUCTION	43
VALIDATION OBJECTIVES FOR THE PERIOD	44
VALIDATION WORK	45
VALIDATION PREPARATION	45
JENKINS PIPELINE BUILD	45
FLAKY TEST HANDLING	46
GLOBAL TEST COVERAGE	47
COMPARING CLOVER REPORTS	48
XWiki DOCKER IMAGES	48
JUNIT5 CONVERSION	49



VALIDATION RESULT	49
PITest/DESCARTES	49
DSPOT	50
CONFIGURATION TESTING	51
EvoCRASH	52
QUALITY MODEL	53
USE CASE : OW2	53
INTRODUCTION	53
VALIDATION OBJECTIVES FOR THE PERIOD	55
VALIDATION WORK	57
VALIDATION PREPARATION	57
DSPOT	57
DESCARTES	58
EvoCRASH	58
VALIDATION RESULT	58
DSPOT	58
DESCARTES	59
EVOCRASH	60
TOOLS PRESENTATION TO PROJECTS	60
QUALITY MODEL	61
ASSESSMENT SUMMARY AND METRICS	61
DETAILED RECOMMENDATIONS	62
DSPOT	62
DESCARTES	63
CAMP	63
EvoCRASH	63
OTHER RECOMMENDATION	63
APPENDIX : KPIs	65
ACTIVEEON	65
ATOS	66
TELLU	67
XWiki	68
OW2	70
APPENDIX : QUALITY MODEL DESCRIPTION (ISO/IEC 25010)	71



1. Executive Summary

This document covers the validation effort conducted by the different partners on the tools developed in the STAMP project. It summarizes the first phase (M3-M18) of the work conducted under the tasks 5.3, 5.4, 5.5, 5.6, and 5.7 on the different use cases provided by namely Activeeon, ATOS, TellU, XWiki, and OW2. With the help of the other partners, these use case providers have applied the STAMP tools according to the “Validation Roadmark and framework V1” described in deliverable D5.2 [4].

2. Revision History

Date	Version	Author(s)	Comments
28-May-18	0.1	Pascal Urso (Activeeon), Mael Audren (Activeeon), Jesús Gorroñogoitia (ATOS), Fernando Mendez (ATOS), Lars Thomas Boye (TellU), Vincent Massol (XWiki), Assad Montasser (OW2)	Initial version for review
22-Jun-18	1.0	Pascal Urso (Activeeon), Mael Audren (Activeeon), Lars Thomas Boye (TellU), Assad Montasser (OW2), Cédric Thomas (OW2)	Final release after addressing peer-review recommendations
22-Jun-18	1.1	Caroline Landry (INRIA)	Adding revision history section

3. Objectives

As expressed in the STAMP “Validation Roadmark and framework”, the objectives of validation activity conducted in the first phase of the project are :

- Provide an early assessment of the functionality and usability of the STAMP techniques and the tool set;
- Provide an initial empirical assessment on the practical use of the early and initial prototypes of the STAMP toolset;
- Involve end-users in the development process of the STAMP toolset, aiming at aligning the toolset features and usage to their industrial needs.

4. Introduction

The main goal of WP5 is to assess that STAMP outcomes address relevant needs in software development industry. The WP5 is particularly in charge to establish the industrial requirements and metrics for validation, to define the validation roadmap and framework, as well as to conduct the actual validation activities. The previous deliverable D5.1 and D5.2 determined the context in which the validation activities are conducted. This deliverable (D5.5 “Use cases validation report v1”) reports the validation activities conducted from month 1 to 18.



Five STAMP partners, namely ActiveEon, ATOS, TellU, XWiki and OW2, are software editors and are committed to evaluate the STAMP results on industrial-grade software they are actively developing and maintaining. This evaluation provides direct feedback to the scientific and technical development conducted in WP1-WP4. The partners conducting validation activities follow different software engineering processes and different requirements and expectations about the STAMP outcomes. This diversity ensures a large coverage for validation of the STAMP outcomes. However, a common quality model and validation framework is established to ensure consistency of validation activities.

This deliverable presents first the global quality model used to evaluate the STAMP tools, then the validation activities are presented per partner. For each partner, we briefly remind the context and specific requirements of the software developed, and the specific validation objectives for this period (M1-18). Then, we report on the validation preparation phase that allowed us to evaluate the usability and accessibility of the STAMP tools. Next, we report in detail on the validation results and quality assessment of evaluated STAMP tools. The deliverable concludes with recommendation for STAMP outcomes improvement, and a summary of main findings.

In appendix, we provide the KPI collected in D5.1, which we augmented with an updated series of measurement.

5. References

- [1] [Likert, Rensis](#) (1932). "A Technique for the Measurement of Attitudes". *Archives of Psychology*. **140**: 1–55.
- [2] ISO/IEC 25010:2011, "Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models", 2011
<http://iso25000.com/index.php/en/iso-25000-standards/iso-25010>
- [3] D5.1 Industrial requirements and metrics v1. C.J. DeLisle et al. STAMP report. 2017:
[d51_industrial_requirements_and_metrics.pdf](#)
- [4] D5.2 Validation Roadmap and framework v1. J. Gorroñoigoitia. STAMP report. 2017:
[d52_validation_roadmap_and_framework.pdf](#)
- [5] D5.5 Use Cases Validation Report V1: [d55_uc_validation_report_v1.pdf](#)

A link to the most recent version of this document.

6. Global Quality Model

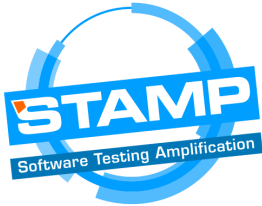
In deliverable 5.2 "Validation roadmap and framework" [4] we have defined a quality model for STAMP tools which is influenced by ISO/IEC 25010:2011. In this standard, the product quality of a software system is evaluated according to eight characteristics with thirty-one sub-characteristics. Not all these characteristics are relevant to STAMP tools or cannot be evaluated at the actual level of maturity of these tools. So we selected six characteristics with ten sub-characteristics, and each use case partner is in charge to evaluate the STAMP tools used during their validation activities according these characteristics.

The list of selected characteristics and their definitions is presented in Appendix. The result of the evaluation is presented in each use case partner section.

7. Use Case : ProActive Workflows and Scheduling (ActiveEon)

Introduction

ProActive Workflows & Scheduling (PWS) supports the execution of jobs and business applications, the monitoring of activity and the access to job results. Allowing IT to scale up and down according to the workload, it optimizes the match between availability and cost. It ensures more work done with fewer resources, managing heterogeneous platforms and multiple sites with advanced usage policies. PWS is a



generic system, used in many different domains. Consequently, it requires a lot of tests on different configurations. STAMP provides the tools that will enable to reach the level of test required for a more stable product.

PWS is a distributed system built upon a microservice architecture pattern. PWS is developed and maintained by a team of 15 developers.¹ The primary software language used to develop the backend is Java, and for the frontend, javascript. For continuous integration, we use Jenkins to launch the jobs that build and test PWS. 150 jobs are defined in the ActiveEon Jenkins server in order to manage the whole product life cycle. 62 jenkins jobs are gathered in one view called “The general ProActive monitor”. A screen in the main developers room of ActiveEon permanently displays this view. If one job of the general ProActive monitor view fails, it becomes a priority for developers to fix it. Every workday a developer is chosen as a “Sheriff” and his role is to ensure that every job in this view is always successful. We use Gradle for build automation. In order to keep a high level of code writing quality, we use the Spotless Gradle plugin² to check and correct the formatting before building. We use SonarQube for continuous inspection of code quality. We also use the dependency management Gradle plugin³ to centralize dependencies in order to avoid conflicts between services.

PWS life cycle is organised through several steps :

Every day, the master branch is built with new features and bug fixes. Once the master branch is built, it is deployed on two servers. One server operates Ubuntu 16.04 and is used for automated system testing. The other one is called “TryDev”, is a Ubuntu 15.10 server, and is used by ActiveEon to develop and test new features.

A release candidate (rc) is created when it is decided to include the new features into a new official release and when all the integration tests pass. About one official release per month is produced. Then, the *rc-release* Jenkins jobs is launched in order to create a release branch from the master branch. The release branch source code is automatically compiled and deployed on the quality assurance platform called “TryQA” which use Debian 8 as operating system. A set of 355 manual tests are performed before the release. A majority of these tests are graphical user interface tests and are performed on the quality assurance platform. Some of those manual tests are configuration testing, and as a consequence the developers may download the release and deploy it on a virtual machine, or in their own computer to execute a manual test. Developers work on the release branch until all the tests pass. The operating system considered during the manual tests are Linux, MacOS and Windows. When a test fail due to a bug, developers produce a bug fix which is committed in the release branch (BF1 in the below figure), and then integrated in the master branch.

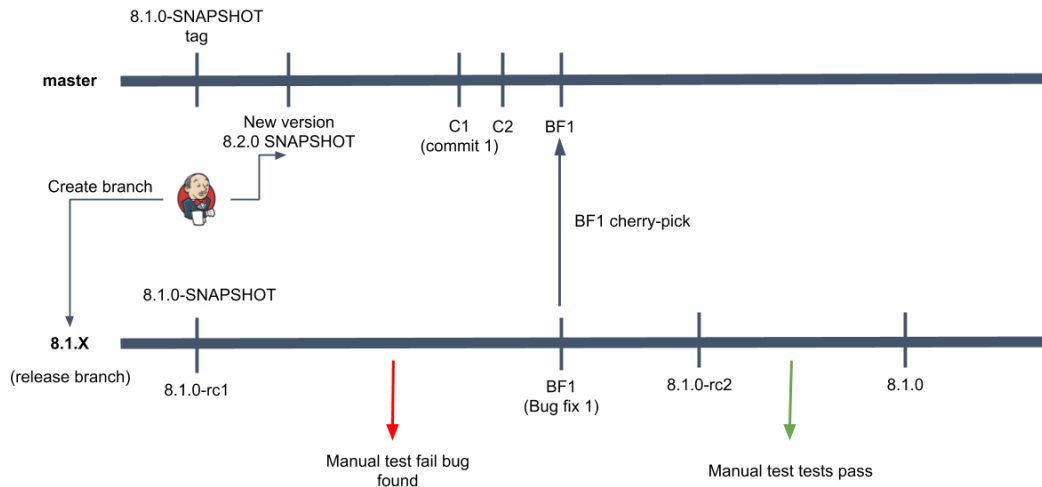
When all the tests pass, the last step to launch the release to tag the current commit on the release branch as the new version (e.g. 8.1.0 in the below figure). This new version is given available for download, and deployed on our publicly available platform called “Try”.⁴

¹ <https://github.com/orgs/ow2-proactive/people>

² <https://github.com/diffplug/spotless>

³ <https://github.com/spring-gradle-plugins/dependency-management-plugin>

⁴ <https://try.activeeon.com/>



Expectations for STAMP tools

Configuration testing of PWS is actually a time-consuming process, which is only achieved on a limited subset of configuration. PWS configuration testing is handled by Jenkins and is limited by the number of Jenkins slaves. CAMP will be used to test PWS on different OS platforms and with different databases improving a lot ActiveEon configuration testing capability by creating dedicated containers.

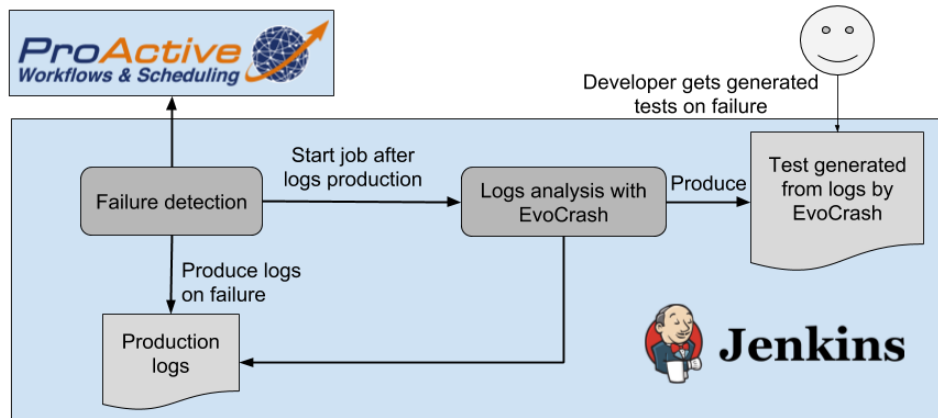
Customer support is an important activity of Activeeon developers. Bugs and exceptions are reported regularly by our consumers. Identifying the origin and the exact contexts of these bugs is time-consuming and could induce delay in bug fixing due to exchanges with consumers. EvoCrash will help to reproduce customer issues and reduce investigation time. Also, the corresponding non-regression tests will be automatically produced. Moreover, thanks to our testing platforms (TryDev, TryQA, and Try), we have direct access (without requiring authorisation or information from our consumers) to production logs. Using EvoCrash on such logs will allow to detect and prevent bug in a proactive way.

Validation objectives for the period

ActiveEon focuses on using two STAMP tools : CAMP for configuration testing and EvoCrash for online testing. Indeed those tools will be the most relevant to fill the needs for PWS quality insurance. At the beginning of the project, Activeeon also tried DSpot in order to give some feedback to partners involved in its development.

EvoCrash

PWS is a project developed during several years, and is composed of 14 microservices. Scheduling is the core service and is separated in four modules. Due to the complexity of the software architecture, new developers usually needs help to understand how it works. Only one person has more than five years experience with the software and worked on all its modules. Five people have two years experience with the scheduling project and have the knowledge about half of its modules. In order to identify the cause and to reproduce the bug when an issue is discovered, it is usual that the developer doesn't know several modules implied in that bug. As consequence it becomes necessary for the developer to ask help or even to reverse engineer a part of the application. The goal is to use EvoCrash to automatically generate tests that reproduce the issue. It will save time avoiding spending time finding where the issue comes from. Moreover after fixing the bug the test can be included in the test suit to validate the fix and to avoid regression in the future. The following schema show the planned architecture where Jenkins and EvoCrash will be combined to generate test on failure.



CAMP

Using CAMP for configuration testing will improve the reliability of our product through the automatic generation of multiple configurations (assembly of different platforms and databases) and the automatic execution of the PWS test suite on these configurations. In our current process, PWS is tested with Java 8 under Debian 8, Ubuntu 15.10, Ubuntu 16.04 and Windows 10 with HyperSQL database (HSQLBD). We use the last versions of Java and Windows, but some of our consumers may use older ones. The HyperSQL database was chosen for efficiency reasons and low impact on our computing resources during continuous integration, but is not frequently used by our consumers. The linux distributions were chosen quite fortuitously, but unify them will only reduce our coverage.

Anyway due to a lack of automation, all the tests are not done under each platform. Moreover it is hard to upgrade the version or to modify the distributions version without breaking the deployment process.

Experience with users has shown that most of PWS issues are linked to the environment. CAMP will improve the test coverage by increasing the number of configuration tested. SO, The ActiveEon's primary goal with CAMP is to detect before the production phase the environment issues and to fix them faster (as those tests should run on a nightly base).

Another goal is for Activeeon to reuse CAMP and provide it as a ProActive Cloud Automation Service, as part of our Automation Dashboard suite of tools.

Validation work

Validation preparation

DSpot

We apply DSpot readme file from <https://github.com/STAMP-project/dspot/blob/master/README.md> in order to ensure that it was compatible with our project.

PWS is a Gradle project and Gradle was not supported at that time. A conversion to a Maven project was attempted in order to match with the Maven format required at that time.

EvoCrash

EvoCrash takes as input the stack trace produced when an exception occurs in a Java software and the source code that produced this exception. We then need to gather PWS production logs, and to adapt them in a format that match EvoCrash required format.

Before STAMP project, all the existing PWS microservice logs were merged in one unique log file. In order to allow EvoCrash to match the exception and the source code of a given service, we first modify all our logging instructions to enforce each of the 14 services to manage its own logging process. Indeed, using a different log file for each service require to harmonize across all the project how the logs were created.



To obtain meaningful production logs, we integrate this refactoring into a PWS version release, that was distributed to our consumers, waiting for them to send issues that contains exploitable logs. This part was longer than expected, since we are not responsible of the adoption speed for the new release of our products, and their issues occurrence.

Eventually, we obtained 15 files containing logs extracted from our consumer production platforms in order to test EvoCrash with different inputs.

CAMP

CAMP tool is use the Docker containerization system to define and execute configuration testing. To be able to use the CAMP configuration testing tool on a given system, one need a Docker file that assemble the image of the system and a parameter file that describe the configuration. Taking inspiration from the XWiki use case, we define a Docker file that download and built the ProActive Workflow and Scheduling Engine.⁵ The parameter file describe a configuration to be tested. The first aim was to test multiple configurations of PWS under different databases.

Validation result

DSpot

During compilation of DSpot under Linux, we discover several compiling issues. Moreover, our use case highlights that DSpot is lacking compatibility with the Gradle dependency manager, more precisely, DSpot does not support multi-modules project. As a consequence PWS is not at this time a relevant project to test DSpot due to this missing requirements. A issue was created on github describing the problem.⁶

EvoCrash

Using EvoCrash highlighted some difficulties to create correct log files as EvoCrash input. The Activeeon testers failed nine times out of ten to transform the PWS logs to an EvoCrash entry. By exchanging with TUD it appears that the log needed to be transformed to meet the EvoCrash format. The transformation steps are missing in the Evocrash-demo documentation and should be added. The exceptions thrown by EvoCrash don't expose the reason why a log parsing failed. An issue was opened in the EvoCrash git project.⁷

```
Target exception was set to: com.fasterxml.jackson.core.JsonParseException LogParser: Failed to parse the log file!  
* Computation finished
```

Eventually, we identified that the log must not be multiparted. For instance, we will delete the "Caused by" section of the following log

```
com.fasterxml.jackson.core.JsonParseException  
    at org.ow2.proactive.scheduler.task.executors.InProcessTaskExecutor.execute(InProcessTaskExecutor.java:225)  
    at org.ow2.proactive.scheduler.task.executors.InProcessTaskExecutor.execute(InProcessTaskExecutor.java:158)  
    ...  
Caused by: org.ow2.proactive.scripting.ScriptException  
    at jdk.nashorn.api.scripting.NashornScriptEngine.throwAsScriptException(NashornScriptEngine.java:470)  
    at jdk.nashorn.api.scripting.NashornScriptEngine.evalImpl(NashornScriptEngine.java:454)  
    ...
```

Also, each stack trace line must end with (<file name>:<line number>). Finally, user need to ensure there is no other log infos, only the stack trace.

⁵ The Docker file is available at the url:

https://gist.github.com/paraita/52854f8b72fff7ea2c96ac2b031bbf13/raw/2205e57f480ba18ba5e9e4dc7be906ab75ed5b3e/scheduler_containering

⁶ <https://github.com/STAMP-project/dspot/issues/95>

⁷ <https://github.com/STAMP-project/EvoCrash-demo/issues/13>

Once the logs are correctly formatted to be used by EvoCrash the other difficulty is to select the good “target frame” parameter value. It is quite hard to find the optimal stack trace depth (“target frame”) to generate desired tests in a reasonable time (less than 1 hour on a developer machine). The only way we found to obtain a working stack trace depth is to set test arbitrary values. Such a process may induce inefficiency and frustration for a developer in an appropriation phase.

Another drawback of the current implementation is the access modifiers related constraints. In our case, EvoCrash fails when the stack trace can only refer to public methods, and EvoCrash fails when using a stack trace that contains a private or protected method.

From a usability point of view, a user needs to create a dedicated java class for a test generator or to call EvoCrash from the command line with a lot of parameters. This constraint makes it hard to integrate the tool in a process because a new class needs to be written, and compiled for each new test.

A good way to improve the test generation would be to script the execution of Evocrash in order to fill automatically the parameters regarding the user needs. Moreover having a generic class for test generation would allow easier target frame parameter selection. User changes could be provided through a property file. This would allow modifications at runtime instead of compilation time.

As a consequence, ActiveEon refactored the Evocrash-demo project in order to ease its usage. A pull request was submitted.⁸ We refactor the project usage by creating a generic class instead of enforcing user to code a dedicated class for each new test generation. This generic class use properties file and no more local class attributes. This change enables to skip the compilation process between two executions. Moreover the project was restructured in order to be used with Gradle. As a consequence, it eases to run the project under any operating system, and embedded in more integrated development environments. The Gradle integration also eases to execute the generated tests. Using Gradle commands, the user can generate the test class that will be compiled and executed in the project.

Here are screenshots that highlight the change for the user to provide the application parameters. On the left we have the previous version, where a Java class must written and where the parameters are provided at compilation time in the java class. On the right we have the new version, using Gradle commands, and where the parameters are provided in a *config.properties* file at runtime.

```

public ProActive_Test() {
    String user_dir = "../";
    File software_dir = new File("/home/michael/Téléchargements/activeeon_enterp
    logPath = Paths.get(user_dir, "resources", "logs", "ProActive", "0", "to_and
    File lib_dir = new File(software_dir, "dist/lib");
    File[] listOffFilesInSourceFolder = lib_dir.listFiles();
    for (int i = 0; i < listOffFilesInSourceFolder.length; i++) {
        String lib_file_name = listOffFilesInSourceFolder[i].getName();
        if (FilenameUtils.getExtension(lib_file_name).equals("jar")) {
            Path depPath = Paths.get(lib_dir.getAbsolutePath(), lib_file_name);
            String dependency = depPath.toString();
            dependencies += (dependency + ";");
        }
    }
    dependencies = dependencies.substring(0, dependencies.length() - 1);
    testPath = Paths.get(user_dir, "GGA-tests");
}

```

To generate tests
`>./gradlew run`

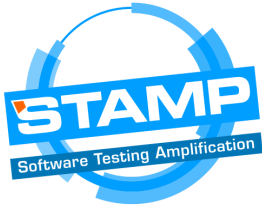
To run tests
`>./gradlew test`

Before (class used to generate tests)

After

From our logs, seven tests were generated, but six of them were generated with an empty test code. Only one has test content and is usable. It seems that the generation is more successful with low stack trace depth values or values that target line of code that belongs to the software and not a library. Here is the begin of a formatted log that generated a non-empty test.

⁸ <https://github.com/STAMP-project/EvoCrash-demo/pull/11>



```
org.objectweb.proactive.extensions.dataspace.exceptions.SpaceNotFoundException: Requested data space is not registered
in spaces directory.
    at org.objectweb.proactive.extensions.dataspace.vfs.VFSSpacesMountManagerImpl.ensureVirtualSpaceIsMounted(VFSSpaces
MountManagerImpl.java:247)
    at org.objectweb.proactive.extensions.dataspace.vfs.VFSSpacesMountManagerImpl.resolveFile(VFSSpacesMountManagerImpl
.java:159)
    at org.objectweb.proactive.extensions.dataspace.vfs.VFSSpacesMountManagerImpl.resolveFile(VFSSpacesMountManagerImpl
.java:129)
    at org.objectweb.proactive.extensions.dataspace.core.DataSpacesImpl.resolveInputOutput(DataSpacesImpl.java:201)
    at org.objectweb.proactive.extensions.dataspace.api.PADDataSpaces.resolveOutput(PADDataSpaces.java:472)
    at org.ow2.proactive.scheduler.task.data.TaskProActiveDataSpaces$1.call(TaskProActiveDataSpaces.java:231)
    at org.ow2.proactive.scheduler.task.data.TaskProActiveDataSpaces$1.call(TaskProActiveDataSpaces.java:228)
    at org.ow2.proactive.scheduler.task.data.TaskProActiveDataSpaces.initDataSpace(TaskProActiveDataSpaces.java:273)
    at org.ow2.proactive.scheduler.task.data.TaskProActiveDataSpaces.initDataSpaces(TaskProActiveDataSpaces.java:228)
    at org.ow2.proactive.scheduler.task.data.TaskProActiveDataSpaces.<init>(TaskProActiveDataSpaces.java:118)
    at org.ow2.proactive.scheduler.task.ProActiveForkedTaskLauncherFactory.createTaskDataSpaces(ProActiveForkedTaskLaunc
herFactory.java:43)
    at org.ow2.proactive.scheduler.task.TaskLauncher.doTask(TaskLauncher.java:184)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:498)
```

In this particular case, the exception comes from a data space property that was not correctly set by our consumer. Data space are remote file systems access (VFS) that host the data shared between the different tasks of the ProActive workflows. Once the log file is formatted correctly, EvoCrash is able to generate tests with three different frame levels: 1, 2 and 3. Selecting a higher frame level prevents EvoCrash to successfully generate a correct test in a measurable time.

CAMP

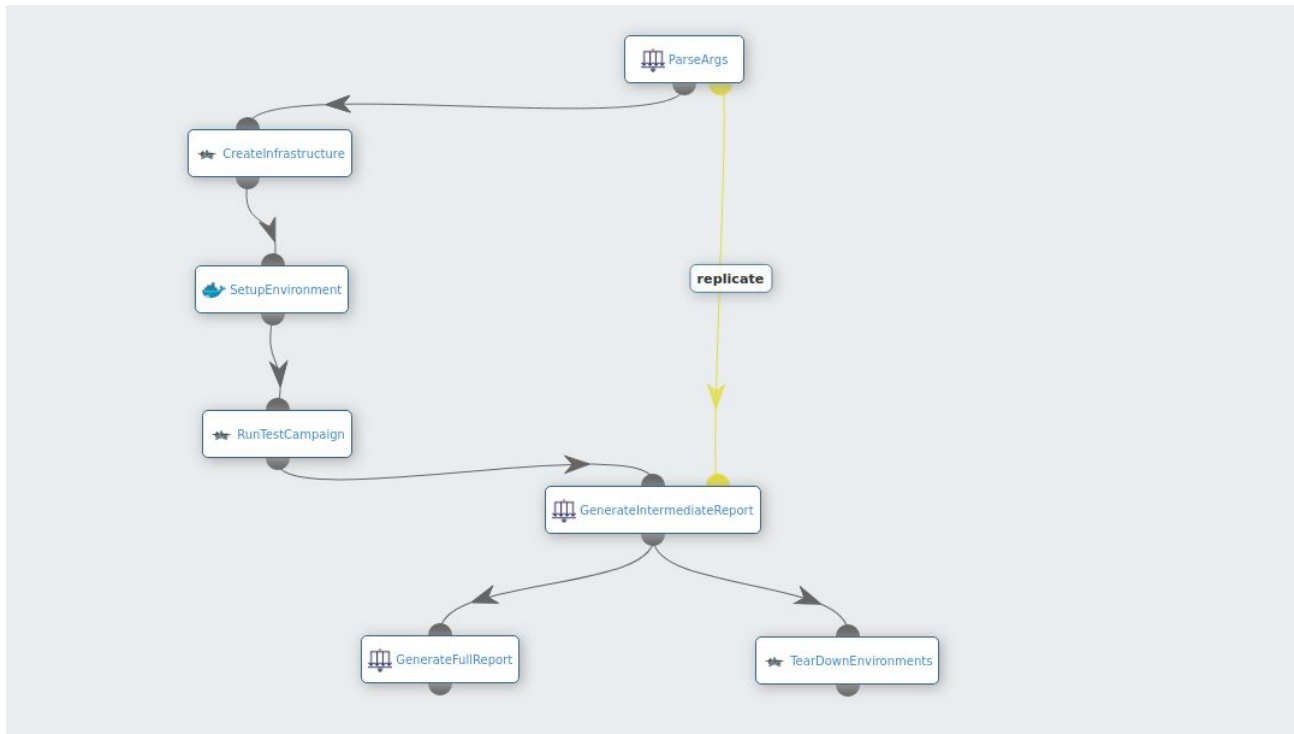
The experimentation of CAMP with PWS first reveals that there is little documentation. In order to use the tool we constructed our configuration starting from an existing one; which is not enough to have a running solution. The first step to improve CAMP would be to increase its usability by improving the documentation with explanation and examples.

In our current CI/CD process, we build a snapshot of our product every night, based on all master branches of all components/services. The result is deployed on two dedicated hosts: one for the dev and another one for the automatic system-tests. The automatic system-tests machine is the an average-sized machine we typically suggest to our potentials or clients willing to test our product on. On this machine, we also use the default settings of our product (HSQLDB, local nodes, defaults JVM parameters). Once the deployment is done, we trigger our system-tests campaign that will target specifically the dedicated system-tests machine through ProActive's REST APIs. Such campaigns last 2 hours and 10 minutes on average. At the end of a campaign, we are strongly confident that we didn't introduce regressions or new bugs for that particular setup (most regressions are caught up here).

We'd like extend this confidence by covering more setups such as:

- MySQL + local nodes
- MySQL + remote nodes using AWS
- PostgreSQL + local nodes
- PostgreSQL + remote nodes using SSH tunneling
- In-Memory Database + local nodes
- In-Memory Database + local nodes + remote nodes

The more combination we test, the better we'll be confident in those setups. To do that, we leverage ProActive's Workflows model to describe the following workflow:



The “ParseArgs” task will parse the CAMP config file and deduce the number of setups to test.

For each setup, we’ll ask our Resource Manager to allocate a new nodesource⁹, for example:

- 1 VM for the PostgreSQL instance
- 1 VM for the ProActive Scheduler instance
- 2 VM for the remote ProActive nodes

The “SetupEnvironment” task will then use this nodesource to deploy the ProActive Scheduler, its nodes and the PostgreSQL by relying on docker-compose. We wrote a Dockerfile¹⁰ that is able to bootrun a snapshot version of our product for CAMP. Last step of this task is to configure ProActive accordingly to the CAMP config file, for this particular setup. The format of that config file is provided by CAMP and should allow us to specify any application-level settings we might need for setting up the infrastructure/environment/product during the setup.

Then we run the system-tests campaign targeting this setup. Once the tests run is finished, we generate and intermediate report so we can start analyzing it if other setups take more time to execute. At the end we strip down all nodesources and we generate the final test results report at the same time.

Quality Model

Characteristic	Sub-characteristic	DSpot	Descartes	CAMP	Evocrash
Functional Suitability	Functional Completeness	4		1	5

⁹ A nodesource is a virtual machine where a ProActive agent is running.

¹⁰ <https://gist.github.com/paraita/52854f8b72ff7ea2c96ac2b031bbf13>

	Functional Correctness	4			1
	Functional Appropriateness	2		3	1
Compatibility	Co-existence	2		4	5
Performance Efficiency	Time-behavior				2
Usability	Appropriateness recognisability	5		1	5
	Learnability	5		1	2
	Operability	2		1	2
Reliability	Maturity	3		1	2
Portability	Installability	4		5	3

8. Use Case : FIWARE Ecosystem (ATOS)

Introduction

Validation Context

Atos Research and Innovation (ARI) department participates in R&D projects funded by different public funding programs, in particular EC H2020. Within these projects, ARI collaborates on the development of innovative software technologies and tools. Internally, ARI Innovation Board selects some of these tools and promotes them internally to become commercial products to be added to the Atos portfolio.

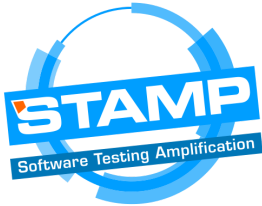
In the context of STAMP tool-sets from two R&D projects have been selected for experimentation with the STAMP test amplification toolset, namely SMART-FI and SUPERSEDE. In SMART-FI, Atos is developing CityGo. In SUPERSEDE, Atos is developing DAPPLE. Both toolsets are currently under evaluation by the ARI Innovation Board.

CityGo is FIWARE IoT-based platform that offers a Web Portal and Android App that citizens can use to obtain the best estimated transport route inside a city by combining several parameters such as user requirements, real time traffic data, location of vehicles, cars and bicycles parking availability and environmental data (weather forecast, social events in the city, etc.). The Web portal provides an informative dashboard that shows statistical data about the human and vehicle traffic inside the city.

CityGo was conceived to address some benefits for the citizens and the city :

- Traffic optimization inside the city
- Reducing the air pollution by encouraging people to use non motorized transportation.
- Low latencies and optimum workload for the transportation services
- Increasing the demand to use public transportation
- Introducing real time city data for the municipality to take actions based on the extreme situations, etc.

An instance of CityGO is currently operating on Malaga city (Spain), integrated with several sensors and data sources (traffic, weather forecast, social events, vehicle and bicycle stations etc..) provided by the Malaga Municipality.



As a solid product, the delivery of CityGO platform in other EU cities (e.g. Geneva, Nice and Toulouse) is under commercial action as these cities have showed their interest on this solution.

CityGo is implemented in different technologies: Python and Angular for the backend and Web Portal, and Java for the Android App. Adopted FIWARE backend services, such as Cygnus or Orion are developed in C++ and Java. Docker and Ansible containerized configurations for installing the CityGo platform are available.

SUPERSEDE develops a platform that assist software stakeholders (e.g. product owners, system architects and administrators, etc) on the software evolution and adaptation for their software systems, based on gathered end-user feedback and system monitoring data. What concerns dynamic software adaptation (i.e. self-adaptation), SUPERSEDE develops a implementation of the MAPE-K control loop. Monitoring is supported by end-user feedback gathering and system monitoring techniques. Analysis uses Big Data techniques to cross and analyse data from these multiple sources, in order to propagate observations on aggregated data. Planning implements decision making methods, based on GAs, that compute optimal configurations (as feature configurations) of the target system that will cope with runtime observations. Execution uses UML models@runtime to keep in sync the target adaptive system configuration. Optimal configurations are injected into the models using AOM techniques. Pluggable enactors interpret these UML models, generates and activates new target adaptive system configurations.

The SUPERSEDE platform is aggregated by an Integration Framework (IF), developed within the project, that provides client-service communication among SUPERSEDE tools, using the WSO2 Integration Platform (ESB, IS, DSS, MB). Docker-containerized configurations for installing the platform are available.

The SUPERSEDE backend components have been developed in different languages, mostly using Java, but also Ruby and Javascript (NodeJS). Front-end components have been developed with JS/HTML.

Atos objectives for STAMP

Software Quality Assurance (QA) is an essential part of Atos IT industrial development life cycle, even within ARI in the context of R&D projects funded by the public administrations. In the last lustrum, in ARI different development teams have been progressively adopting agile methodologies for development and DevOps pipelines for CI/CD. Testing activities have been incorporated within these practices although not as widely as on the commercial development projects.

STAMP gives ARI a great opportunity to promote *testing* as an essential activity in the process development lifecycle and the DevOps pipeline in the development of R&D projects, with the adoption of testing best practices and test amplification techniques, which eventually can be promoted to commercial IT projects.

The ARI teams that are working on the development of the two described UC, based on the QA experiences acquired within them, have identified key motivating challenges for the adoption of the STAMP test amplification techniques, aiming at improving the QA process itself and its results.

Tests Suites are designed and iteratively implemented and conducted during the development of these projects. However, the efficacy of these test suites to anticipate the detection of issues before delivery is largely improvable. Best practices for test planning and engineering need to be adopted, in order to cope with several test situations have been faced in Atos cases, including:

- Tests pass despite some runtime exceptions are thrown;
- Tests setups are not appropriate, particularly when the system-under-test is not set to the correct precondition state;
- Tests assertions are not always correct or complete;
- Test coverage is low, hence the complete system software is not adequately tested;
- Regression bugs are not detected by test suites, so they pop up among releases;
- We are suffering flaky tests because of race and environmental conditions (i.e. network cutoffs, unavailability of test input resources, etc);
- The analysis and reproduction of runtime failures are time-consuming and a human-specific task, where automation seems not to be possible;

- Testing maintenance is costly. Inadequate maintenance increases the occurrence of flaky tests, false positives, hiding of regression bugs, etc.

Test execution also faces a number of challenges detected in the context of Atos UCs. Test preparation and mocking requires significant engineering work and maintenance. As most of test suites available in Atos UCs are integration and system tests, the delivery and configuration of the system-under-test requires complex DevOps pipeline for CI/CD. This is particularly problematic when executing tests locally (by individual developers) even considering the availability of a remote system-under-test. Moreover, the eventual future commercial distribution of CityGo will require the conduction of system and integration assessment in different configurations of the system-under-test, in particular:

- Configurations that modify the baseline dependencies of the system-under-test, including OS, databases, programming/execution frameworks, application containers, and other dependencies. Assessing the functional equivalence of these configurations (i.e. successful passing of sanity checks) is essential for the delivery of CityGo to new cities. These new configurations should embrace the technology baseline imposed by the infrastructure of the hosting city.
- Configurations that improve the capacity of CityGo backend and front-end (Android App, Web Portal) w.r.t. some non-functional properties (e.g. performance, resources consumption, availability, etc). This ability is required to tune CityGo parameterization to the execution needs (e.g. number of users, average workload per user) on the different CityGo instantiations.

STAMP test amplification techniques will be adopted to cope with above challenges and objectives. The results of this adoption will be reporting on all the reports of this series.

Validation work

This report describes the validation-related activities conducted by Atos in the period M6-M18 (April 2017-May 2018).

Quality Model

As mentioned in D5.2, STAMP toolset are evaluated with the ultimate purpose of assessing whether from the usage of this toolset can be obtained tangible benefits in relation to the industrial Atos objectives summarized in previous section. In order to qualify and quantify the fulfilment of these expectations, we select a subset of quality attributes or characteristics from standard quality models for QA, such as the ISO 25010 quality model.

Considering that during the assessment period reported in this document, M6-M18, STAMP toolset is under development, so released toolset is immature, the subset of quality model characteristics/sub-characteristics Atos has focused on are the following:

Product Quality:

Characteristic	Sub-characteristic	Definition
Usability	Appropriateness recognisability	degree to which users can recognize whether a product or system is appropriate for their needs.
	Learnability	degree to which a product or system enables the user to learn how to use it with effectiveness, efficiency in emergency situations.
	Operability	degree to which a product or system is easy to operate, control and appropriate to use.

Table T1: Product quality model adopted in Atos UC validation

Quality In Use:

Characteristic	Sub-characteristic	Definition
Efficiency		resources expended in relation to the accuracy and completeness with which users achieve goals
Satisfaction	Usefulness	degree to which a user is satisfied with their perceived achievement of pragmatic goals, including the results of use and the consequences of use

Table T2: Quality in use model adopted in Atos UC validation

Besides these quality model characteristics, Atos validation in M6-M18 period has adopted and computed some of the KPIs for STAMP defined in D5.1. Appendix 14 collects the KPI metrics collected in Atos UCs in order to assess progression.

Validation objectives

Atos validation objectives for this assessment period can be specialized by UC:

CityGo

The primary objective of CityGo UC is to generate new testing configurations based on Docker and Ansible deployment descriptors. These descriptors can be parameterized w.r.t. two main objectives:

- obtain functionally-compatible configurations of baseline requirements, such as OS, databases, development/execution environments, application containers, Web engines, etc;
- obtain optimal configurations that maximize/minimize some measurable non-functional properties, such as performance, throughput, reliability, resources consumption, etc.

Once the set of amplify test configurations are generated, CityGo software can be tested on them. These configurations are used to instantiate different instances of CityGo using Docker/Ansible container engine. Once instantiated, these instances are the system-under-test for:

- running sanity-checks that assess the functional equivalence and validity of each CityGo instance;
- running stress-tests that assess the optimization of the non-functional properties.

A secondary objective is to increase the test coverage for the FIWARE GE services that are included into the CityGo backend, namely Orion Context Broker and Cygnus.

SUPERSEDE

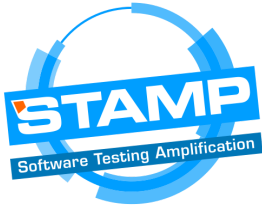
The primary objective of SUPERSEDE UC is to improve the efficacy of the test suites that accompany the source code. By efficacy we mean the ability of test suites to:

- execute a wider set of lines of code and execution branches: increase the test coverage;
- test the source code under a wider range of valid input sets: increase the number of tests and assertions;
- detect more execution deviations from expected results: increase the number of bugs detected;
- detect more execution exceptions, including crashes: increase the number of bugs detected;
- determine the quality of existing test suites: detect tests that are not capable to detect bugs in the code they cover;

Validation preparation

Atos assessment during this reporting period will target the evaluation of the following STAMP tools:

- DSpot: <https://github.com/STAMP-project/dspot>
- Descartes: <https://github.com/STAMP-project/pitest-descartes>



- CAMP: <https://github.com/STAMP-project/camp>

STAMP Tool	Version Tested	Purpose for Atos UC
DSpot	Latest build in GitHub: 1.1.1-SNAPSHOT	Amplify the number of test. Increase test coverage and assertions
Descartes	1.1	Determine the quality of test suites
CAMP	Latest build in GitHub	Amplify the test configurations: focus on required dependencies

Table T3: STAMP tools evaluated in Atos UC

The usage of these tools for test amplification requires the preparation of input artifacts. The availability of these artifacts, or the possibility to provide them, together with the compatibility of these tools with existing code base and CI/CD pipelines, determine the UCs and specific components within them, that could be target for test amplification using STAMP tools. The following requirements that must be satisfied, constrained the selection of Atos UCs and components for evaluation:

STAMP Tool	Requirements
DSpot	Maven/Gradle Java projects
Descartes	Maven/Gradle Java projects
CAMP	Docker descriptors for component containerisation and composition

Table T4: Requirements for using STAMP tools

Based on this requirements and constrains, the following considerations were made to select target Atos UC components for test amplification evaluation:

CityGo:

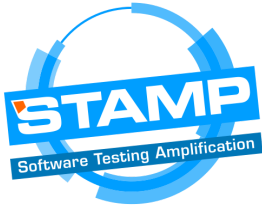
CityGo CI/CD did not support Docker containerisation at the beginning of the evaluation process, as requested by CAMP tool, but it was planned as an important improvement. Indeed, some FIWARE GE services adopted by CityGo already supported Docker containerisation. Hence, providing Docker support for CityGo only required to extend this support to the other backend components (Apache, PostgreSQL, Python, Angular). As Atos was interested to decouple the test configuration amplification technology form the baseline containerisation technology, it was also planned to support the CI/CD of CityGo using Ansible, a competitive technology for software delivery.

Most of CityGo components are developed in programming languages not compatible with DSpot/Dcartes tools. Only one of the FIWARE GE services included in the CityGo backend, Cygnus, an event dispatcher for IoT is implemented in Java and managed by Maven. Therefore, the evaluation of unit test amplification in CityGo UC will target this component.

SUPERSEDE:

The SUPERSEDE UC focuses on two Java based components:

- DAPPLE: an enactment (execution) platform for dynamic adaptation, developed in Java.
- Integration Framework (IF): a interoperability library for component communication, developed in Java.



However, the following limitations were detected that hamper somehow the usage of DSpot/Descartes tools for unit test amplification:

- DAPPLE: few DAPPLE components are managed by Maven/Gradle. This is so because most of the components are Eclipse plugins, standalone executed, but managed by Eclipse MANIFEST. Only the projects that wrap an aggregation of these components as services are managed with Gradle. The Maven technology used to support the integration between Gradle/Maven and Eclipse plugins, Tycho, works well with Maven, but produces in-memory POM descriptors. This is the only existing approach to combine Eclipse plugins with Maven, but unfortunately is not compatible with DSpot/Descartes tools. Therefore, existing DAPPLE test cases in Eclipse plugins cannot be directly amplified with STAMP tools. Only those in the few Gradle projects that wrap the plugins as services. Additionally, at the time of evaluating, Gradle support for DSpot/Descartes was not complete¹¹, so it was required to create Maven support for these projects. Despite these limitations, Atos is quite interested in improving test suites for DAPPLE platform, so further investigation will clarify strategies to adapt DAPPLE code and test base to DSpot/Descartes constraints.
- IF: it is developed in Java and managed by Gradle. As commented above for DAPPLE, the lack of Gradle support in DSpot/Descartes at the time of starting the evaluation required to create Maven support for IF.

During these evaluation period, Atos has been working on preparing the following artifacts:

DSpot/Descartes

Experimentation scripts

We created Bash scripts (for Linux) that automated the execution of DSpot and Descartes on selected CityGo and SUPERSEDE test suites, collecting execution statistics and logs, and organizing the test amplification validation experiments by date and configuration. These scripts for IF evaluation are available at:

DSpot:

<https://github.com/supersede-project/integration/blob/master/descartes/IF/API/eu.supersede.if.api.executeDSpot.sh>

Descartes:

<https://github.com/supersede-project/integration/blob/master/descartes/IF/API/eu.supersede.if.api.executeDescartes.sh>

These scripts required others located at the same folder: record_stats_repeat.sh, record_stats.sh

Maven descriptors

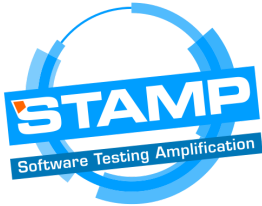
As commented, a Maven POM descriptor from Gradle configuration was created for SUPERSEDE IF test amplification.

Green test suites

Descartes/DSpot tools require green test suites (i.e. all test suites pass) before they can be amplified. As most of SUPERSEDE IF test suites are integration test, with complex test setups that are quite sensitive to the external state of some services, they are prone to fail when these conditions are not hold. Therefore, before assessing the test amplification using DSpot/Descartes we need to guarantee all target test suites pass. To do so, we expend resources to:

- Improve the initial quality of test suites, in particular the test preparation (@BeforeClass);

¹¹ Gradle support for DSpot/Descartes is under development and it will be evaluated in next evaluation period.



- Create helper classes that reset the system-under-test (i.e. SUPERSEDE backend) to the state required for testing;
- Disable (@Ignore) problematic tests: those whose show difficulties to reset the testing state to a valid one.
- Configure <targetTests> Descartes/PITest configurations in pom.xml

Experimentation process

Target UC: DSpot and Descartes tools have been used to amplify test suites of SUPERSEDE IF component.

Initial Tool setup:

- Descartes: version 1.1 was obtained from GitHub repository:
`git clone https://github.com/STAMP-project/pitest-descartes.git -b descartes-1.1`
- DSpot: latest snapshot version was obtained from GitHub repository:
`git clone https://github.com/STAMP-project/dspot.git`

Experimentation setup: experiments were conducted in a server, Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz, 6 cores, 32 Gb RAM. This is not a dedicated server, also the services of the system-under-test are coexisting within. Therefore, we cannot make conclusions on the measured performance for test amplification. Next conducted experiments will use a dedicated server for test amplification process, detached from the server that hosts the system-under-test.

In the experimentation server we installed:

- The STAMP tools target of validation: DSpot and Descartes;
- The SUPERSEDE IF and test suites: the test suites to be amplified;
- The SUPERSEDE platform: the system-under-test that IF test suites test.

Initial experiments: Initial experiments aiming at familiarizing with Descartes and DSpot tools were manually conducted in a non-systematic manner, consulting available documentation at:

Descartes: <https://github.com/STAMP-project/pitest-descartes/blob/master/README.md>

DSpot: <https://github.com/STAMP-project/dspot/blob/master/README.md>

but also supported by the tools developers through their issue tracker:

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

DSpot: <https://github.com/STAMP-project/dspot/issues>

The objective was to ensure a correct configuration and usage of the tools. As a result of this initial experiments, a number of tickets (for reporting bugs, correcting misuses or requesting support) were issued in their trackers.

Systematic experiments:

DSpot: experiments have been conducted using the script `executeDspot.sh`. This script executes DSpot in selected SUPERSEDE IF test suites with a given DSpot execution configuration. It stores execution results in the `dspot-usecases-output` repository, which is manually push to Github. Execution results are organized according to this folder structure:

```
atos/<use_case>/<dspot_selector>/<dspot_amplifiers>/<experiment_date_time>/
```

where <use_case> is supersede, <dspot_selector> is the selector included in the DSpot configuration (e.g. Clover), and <dspot_amplifiers> is a colon separate list of amplifiers included in the DSpot configuration,



Each experimentation folder includes:

- DSpot execution logs
- Execution monitoring data, including CPU and Memory consumption, sampled periodically
- experimentation configuration: reported at the beginning of the execution logs
- experimentation results: amplified tests, textual and JSON report.

Descartes: similarly to DSpot, Descartes experiments are launched using `executeDescartes.sh` script, which executes Descartes on selected IF test suites, specified in the `pom_for_descartes.xml` configuration. Execution results are stored in `descartes-usecases-output` repository, pushed to Github. Execution results are organized according to this folder structure:

`atos/<use_case>/<experiment_date_time>/`

Each experimentation folder includes:

- Descartes execution logs
- Execution monitoring data, including CPU and Memory consumption, sampled periodically
- experimentation configuration: `pom_for_descartes.xml`
- experimentation results: PITest HTML and JSON report.

Reporting process

Results of experimentation with DSpot and Descartes are provided to the developers through different channels and formats:

- Issuing tickets in GitHub tracker, reporting issues such as potential bugs, enhancements (new features), requests for support, etc.
- Posting experimentation results in GitHub repositories: `dspot-usecases-output` and `descartes-usecases-output`
- Posting messages by email.
- Reporting in WP5 deliverables like this document.

CAMP:

Test configuration descriptors

The usage of CAMP tool in Atos CityGo for amplifying test configurations, requires the existence of test configurations for deployment, which will be amplified. During this evaluation period, Atos has focused on:

- Creating deployment descriptors for CityGo components, based on Docker and Ansible technologies
- Parameterizing these descriptors for supporting test configuration amplification with optimal parameterization sets.

Docker and Ansible descriptors of CityGo components are available at the STAMP GitLab:

<https://gitlab.ow2.org/stamp/atos-uc-city2go>

Figure F1 represents the distribution of CityGo components and dependencies:

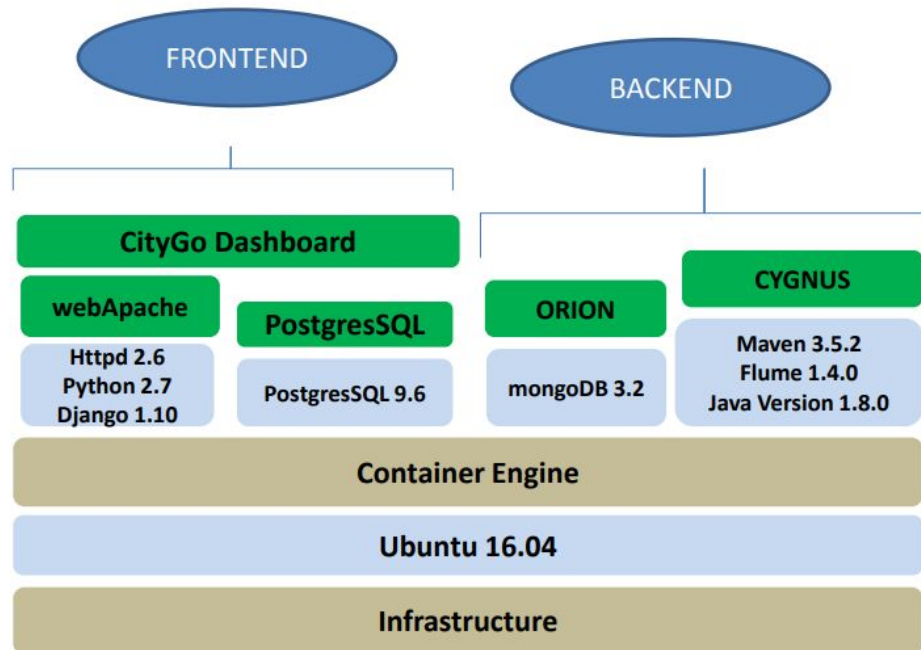


Figure F1: CityGo components and dependencies

Docker-based deployment configuration

Docker descriptors for CityGo are available at:

<https://gitlab.ow2.org/stamp/atos-uc-city2go/tree/master/Descriptors/Docker>

They consist of:

- Dockerfiles for building container images for some CityGo components: PostgreSQL and CityGo Dashboard (Apache, Python, Angular). FIWARE already provided Docker images for the other backend components: Cygnus and Orion.
- A parameterizable Docker Compose descriptor that compose up the entire CityGo platform, as an aggregation of interlinked containers
- Configuration templates for some CityGo components: Apache, PostgreSQL.

The parameterization of CityGo components is technology specific. In particular, Apache is optimized through its MPM Worker¹² or Prefork¹³ module configurations. The concrete parameterization set is defined in the *docker-compose.yml* configuration (Figure F2). Then, Docker Compose injects this configuration into the container of the CityGo WebApp. Finally, within the container, the Linux *envsubst* program replaces this parameterization set into the Apache module descriptors. Similarly, PostgreSQL configuration is parameterized (Figure F2).

¹² <https://httpd.apache.org/docs/2.4/mod/worker.html>

¹³ <https://httpd.apache.org/docs/2.4/mod/prefork.html>


```

services:
  db:
    build: ./db
    container_name: my_postgres
    environment:
      - max_connections=500
      - shared_buffers=256
      - port=5432
      - POSTGRES_DB=*****
      - POSTGRES_USER=citygo
      - POSTGRES_PASSWORD=*****
      - PGDATA=/var/lib/postgresql/data/pgdata
    ports:
      - "5432:5432"
    expose:
      - "5432"

  web:
    build: ./webapp/ShowcaseServer
    container_name: my_web
    environment:
      - PYTHONUSERBASE=./webapp/ShowcaseServer/
      - StartServers=2
      - MinSpareThreads=25
      - MaxSpareThreads=75
      - ThreadLimit=64
      - ThreadsPerChild=25
      - MaxRequestWorkers=150
      - MaxConnectionsPerChild=0
      - MinSpareServers=5
      - MaxSpareServers=10
    volumes:
      - ./webapp:/webapp
    ports:
      - "80:80"
    command: python /webapp/ShowcaseServer/manage.py runserver 0.0.0.0:80
    depends_on:
      - db
      - orion

```

Figure F2: Docker Compose parameterization of CityGo services: Apache and PostgreSQL

Ansible-based deployment configuration

Ansible descriptors for CityGo are available at:

<https://gitlab.ow2.org/stamp/atos-uc-city2go/tree/master/Descriptors/Ansible>

They consist of:

- An Ansible descriptor (ansible.yml) that declares the main roles (i.e. services) in CityGo
- A set of Ansible task descriptors for individual roles (main.yml)
- A declaration of predefined variables that are referenced in above descriptors.

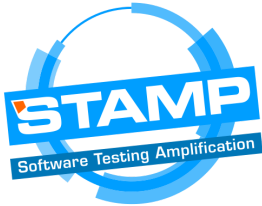
Additional documentation on CityGo experimentation with Docker and Ansible is available at STAMP GitLab repository:

<https://gitlab.ow2.org/stamp/atos-uc-city2go/tree/master/docs>

In order to execute CAMP with CityGo we follow the following procedure:

1. Clone CAMP from Github: <https://github.com/STAMP-project/camp.git>. This will create a *camp* folder.
2. Clone CityGo from Github: https://gitlab.atosresearch.eu/ari/stamp_docker_citygoApp.git. This will create a *stamp_docker_citygoApp* folder.
3. Replace repo folder in *camp/samples/stamp/atos* with the one located at *stamp_docker_citygoApp*
4. In folder *camp/samples/stamp/atos* invoke CAMP with the following command:

```
docker run -it -v $(pwd):/root/workingdir songhui/camp /bin/bash allinone.sh
```

Validation results

Descartes

UC: Supersede. Component: IF

Experiments: Descartes tool was executed on the test suite of:

- the SUPERSEDE IF component, in the experimentation setup described above, using the experimentation method and tooling also described above.

UC: CityGo, Component: fiware - cygnus

Experiments: Descartes tool was executed on the test suite of:

- the CityGo Cygnus component which has got two sub-component, cygnus-common (Apache Flume) and cygnus-ngsi (Agent)

SUPERSEDE Observations: Descartes cannot be applied to the entire Supersede IF test suite, as Descartes/PITest complains that the test suite is not green. However, test suite is green for Maven, as *mvn test* passes all tests excepting those marked with *@Ignore*, which are skipped. Those tests showing problems with Descartes/PITest were marked as ignorable. Despite this solution, Descartes/PITest still complains that the test suite is not green. This issue was reported to Descartes development team (issue #28, #50). The only working solution is to filter out conflictive test suites using PITest *<targetTests>* configuration to include only those that are green. However this alter the overall results as some packages of the IF component are not covered by Descartes execution (i.e. packages with 0% line coverage in below figures).

Citygo Observations: Descartes is applied to the entire of CitiGo test suit, however some test that must be marked with *@Ignore* to skip them in order to obtain green test suite, as commented above. However, all test suites are green for Maven. Another useful working solution is to filter out all conflictive test suites using *<targetTest>* tag setting up in the Descartes configuration in the pom file.

During the evaluation of Descartes, Atos evaluation team reported a total number of 8 issues in the Descartes GitHub tracker.

Results:

- The execution of Descartes on SUPERSEDE IF are located at:

<https://github.com/STAMP-project/descartes-usecases-output/tree/master/atos/supersede/>

Latests and widest (in the same of being applied to the widest possible set of Supersede IF test suites, which is reported and commented hereafter is located at:

https://github.com/STAMP-project/descartes-usecases-output/tree/master/atos/supersede/2018-04-25_16:41:19

- The execution of Descartes on CityGo Cygnus are located at:

<https://github.com/STAMP-project/descartes-usecases-output/tree/master/atos/Citygo>

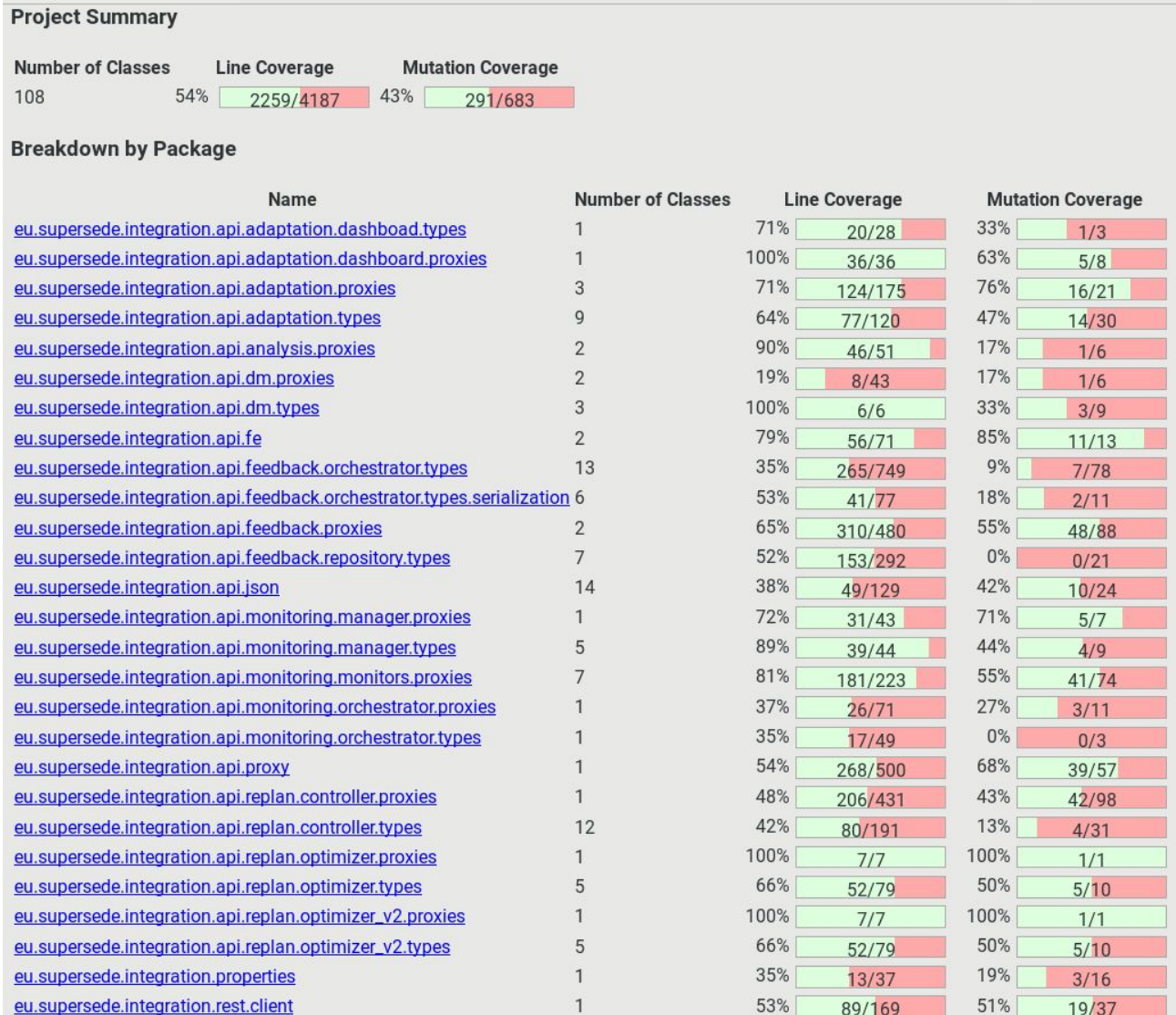
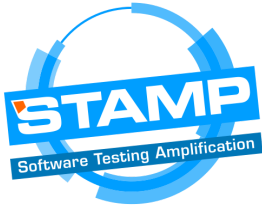


Figure F3: Excerpt of the STAMP Descartes/PITest report for Supersede IF component (I)

The extended Descartes report, in Github markdown format (Figure F4), is available at:

https://github.com/STAMP-project/descartes-usecases-output/blob/master/atos/supersede/2018-04-25_16:41:19/report/index.md



Atos Supersede

Time: 1:57:51.642000

Issues: 89

Classes

- eu.supersede.integration.api.replan.controller.proxies.ReplanControllerProxy **Issues: 5**
 - deleteFeatureByIdOfProjectById(int,int) (partially-tested)
 - deleteProjectById(int) (partially-tested)
 - deleteReleaseByIdOfProjectById(int,int) (partially-tested)
 - deleteResourceByIdOfProjectById(int,int) (partially-tested)
 - deleteSkillByIdOfProjectById(int,int) (partially-tested)
- eu.supersede.integration.api.feedback.proxies.FeedbackOrchestratorProxy **Issues: 5**
 - createUser(eu.supersede.integration.api.feedback.orchestrator.types.User,long) (pseudo-tested)
 - createUserGroup(eu.supersede.integration.api.feedback.orchestrator.types.UserGroup,long) (pseudo-tested)
 - deleteApplication(long) (pseudo-tested)
 - deleteConfiguration(long,long) (pseudo-tested)

Figure F4: Excerpt of Extended Descartes report for SUPERSEDE IF Test Suite

Analysis of results:

Descartes/PITest report is shown in Figure F3. Listing L1 shows a mutation score of 30%: only 278 of 935 mutations were detected by the test suite.

```
=====
- Timings
=====
> scan classpath : < 1 second
> coverage and dependency analysis : 3 minutes and 4 seconds
> build mutation tests : < 1 second
> run mutation analysis : 1 hours, 29 minutes and 57 seconds
-----
> Total : 1 hours, 33 minutes and 2 seconds
-----
=====
- Statistics
=====
>> Generated 683 mutations Killed 291 (43%)
```

```
>> Ran 3459 tests (5.06 tests per mutation)
```

Listing L1: Descartes/PITest report for Supersede IF component

Extended Descartes report (see figure F4) shows 89 issues detected, structured by codebase class. For instance, for class *eu.supersede.integration.api.replan.controller.proxies.ReplanControllerProxy* 5 partially-tested issues (i.e. extreme mutations detected by some tests, but not by others) were detected. As another example, *eu.supersede.integration.api.feedback.proxies.FeedbackOrchestratorProxy* shows 5 pseudo-tested issues (i.e. extreme mutations not detected by tests).

Clicking on an issue provides further details:

- Transformations (i.e. mutations) applied:
 - specifying which ones pass or not the tests, for partially-tested methods
- Associated test classes

This information helps developers to identify the test methods that require improvements to remove the issue, either by refactoring the test methods or by adding additional assertions.

Atos is using this information for improving SUPERSEDE IF test suites, investigating strategies to fix detected issues, based on test method refactoring and oracle improvements. As an example, the analysis of the issues associated to the codebase class *ReplanControllerProxy* indicates its delete methods used in test suites are not verify with assertions. These delete methods were used in test suites to clean up the system-under-test after being modified for other test purposes.

As a result of these test improvement strategies, we expect to significantly increase the mutation score and the code coverage (see KPI section), as well as reducing the number of issues detected by Descartes.

At the time of writing, the IF test suites have been refactored (also the codebase of the SUT) in order to remove most of the issues detected by Descartes that were classified as pseudo-tested method. For partially tested method, particularly those associated to boolean return types, we need to further explore corrective strategies.

New Descartes analysis on the refactored test suites has been conducted, showing the following results (Listing L2):

```
=====
- Statistics
=====
>> Generated 708 mutations Killed 317 (45%)
>> Ran 2569 tests (3.63 tests per mutation)
```

Listing L2: Descartes analysis on the refactored test suites for Supersede IF component

By refactoring the SUPERSEDE IF Test suites for removing pseudo tested methods, we obtained 45% of mutation coverage, compared to the previous value: 43% (this an relative increment of 4.7%). The line coverage shown by PITest report is not increased (54%).

By other hand, in the Listing L3 it is shown the Descartes/PITest report for Citygo FIWARE cygnus-common component.

It shows a mutation score of 32%: only 150 of 465 mutations were detected by the test suite add in not killed mutators, those lines that of code not covered.



```
> Total : 1 minutes and 25 seconds
```

```
-----  
- Statistics  
=====
```

```
>> Generated 465 mutations Killed 150 (32%)  
>> Ran 597 tests (1.28 tests per mutation)
```

Listing L3: Descartes/PITest report for Citygo cygnus-common component

Listing L4 shows the Descartes/PITest report for the Citygo FIWARE cygnus-ngsi component. Equally, it shows a mutation score of 48%, but only 191 of 401 mutations were detected by the test suite.

```
-----  
> Total : 34 seconds  
-----
```

```
=====
```

```
- Statistics  
=====
```

```
>> Generated 401 mutations Killed 191 (48%)  
>> Ran 445 tests (1.11 tests per mutation)
```

Listing L4: Descartes/PITest report for Citygo cygnus-ngsi component

DSpot

UC: Supersede. Component: IF

Experiments: DSpot tool was executed on the test suite of the SUPERSEDE IF component

UC: CityGo, Component: fiware - cygnus

Experiments: DSpot tool was executed on the test suite of the CityGo Cygnus sub-components: cygnus-common and cygnus-ngsi

SUPERSEDE Observations:

DSpot has been experimented in individual test cases with different selectors, mutators and run configurations.

- DSpot is not working with Clover and PITest coverage selectors in SUPERSEDE IF test suites. Related issues have been reported in [#329](#). Default PITest selector has been deprecated, so it won't be used anymore. In the following, all experiments have been conducted with Jacoco coverage selector.
- DSpot works well with some IF test suites but not with others. Detected issues have been reported in [#425](#) and [#426](#). Based on developers' feedback, new experiments will be conducted and results reported.

Citygo Observations:

DSpot has been experimented in the entire test suite of CitiGo and individual test cases, configured to use different selectors (e.g. JacocoCoverageSelector, CloverCoverageSelector) and run configurations. Despite the entire test suite is green for Maven, DSpot crashes during execution. This issue was reported to DSpot development team. ([#424](#))



During the evaluation of DSpot, Atos evaluation team reported a total number of 19 issues in the DSpot GitHub tracker.

Results:

- The execution of DSpot on SUPERSEDE IF are located at:

<https://github.com/STAMP-project/dspot-usecases-output/tree/master/atos/supersede>

This repository is structured according to this schema <selector>/<MutatorA:MutatorB:...>. For instance, experiments with Jacoco selector and mutators: MethodAdd, StatementAdd and TestDataMutator are located at:

<https://github.com/STAMP-project/dspot-usecases-output/tree/master/atos/supersede/JacocoCoverageSelector/MethodAdd:StatementAdd:TestDataMutator>

- The execution of DSpot on CityGo Cygnus are located at:

<https://github.com/STAMP-project/dspot-usecases-output/tree/master/atos/Citygo>

In the experiments DSpot has worked with IF, results shown an increased on the test coverage thanks to the new generated tests. As an example, the amplification of *eu.supersede.integration.api.analysis.proxies.test.** test suite results on 7 amplified tests and an increment of 0.91% on the test coverage (Listing L5).

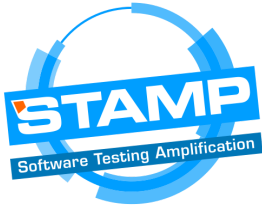
```
===== REPORT =====  
Initial instruction coverage: 303 / 33335  
0.91%  
Amplification results with 7 amplified tests.  
Amplified instruction coverage: 303 / 33335  
0.91%
```

Listing L5: DSpot report for SUPERSEDE IF *eu.supersede.integration.api.analysis.proxies.test.** test suite

Note that this modest result is consequence of the amplification of only one of the available test suites, hence starting with a very low coverage (i.e. 303/33335). Amplifying the entire test suite should show better results. This DSpot run took 37' 52". We have reported about other DSpot runs on single test suites that took more than 12 hours to complete (#386). These long DSpot execution times seem to be related to the fact that IF test suites show long I/O waits as these tests are invoking remote services in the SUPERSEDE microservice infrastructure.

We could generate with DSpot a total of **14 new test suites**, consisting of **575 amplified tests**. Not all these tests were valid, as some of them failed when run again. We discarded those tests, remaining **12 valid new test suites**, consisting of **484 valid tests**.

Then, we computed the test coverage using Clover plugin for Maven with the original test suites, and with the amplified tests suites, obtaining the figures collected in Table T5.



	Number of Tests	Code Coverage
Original Test Suite	143	49.2%
Amplified Test Suite	624	50%

Table T5: Code coverage amplification for IF code base

This modest amplification of code coverage (0.8% absolute, 1,6% relative) considers the complete IF code base. However, DSpot has only amplified few test cases that covers a portion of the total code base. If we consider the code coverage amplification of individual IF packages, figures are better (see Table T6 for some package examples), ranging from 2 % up to 22 %.

Package	Original Code Coverage (%)	Code Coverage after test amplification(%)	Relative Increment (%)
eu.supersede.integration.api.replan.controller.types	37.4	45.8	22.46
eu.supersede.integration.api.replan.controller.proxies	46.6	49.3	5.79
eu.supersede.integration.api.monitoring.monitors.proxies	76.7	78.6	2.47
eu.supersede.integration.api.monitoring.manager.types	84.5	87.9	4.02
eu.supersede.integration.api.analysis.proxies	86.3	88.2	2.2
eu.supersede.integration.api.fe	76.7	81.4	6.13

Table T6: Code coverage amplification for IF code base

CAMP

CAMP generates two additional DockerCompose configurations when it is executed on CityGo application. Each of these configurations provides a different performance configuration set for the Apache web service of CityGo. Default and additional configurations for this service, generated by CAMP, are shown in figure F5.


```
web:
  build: ../repo/Showcase
  image: showcase:default
  container_name: my_web
  environment:
    - PYTHONUSERBASE=../webapp/ShowcaseServer/
    - StartServers=2
    - MinSpareThreads=25
    - MaxSpareThreads=75
    - ThreadLimit=64
    - ThreadsPerChild=25
    - MaxRequestWorkers=150
    - MaxConnectionsPerChild=0
    - MinSpareServers=5
    - MaxSpareServers=10
  volumes:
    - ../webapp:/webapp
```

```
web:
  build: ../repo/Showcase/ShowcaseServer
  command: python manage.py runserver 0.0.0.0:80
  container_name: my_web
  depends_on:
    - db
    - orion
  environment:
    - PYTHONUSERBASE=../webapp/ShowcaseServer/
    - StartServers=2
    - MinSpareThreads=25
    - MaxSpareThreads=75
    - ThreadLimit=128
    - ThreadsPerChild=100
    - MaxRequestWorkers=150
    - MaxConnectionsPerChild=0
    - MinSpareServers=5
    - MaxSpareServers=10
  image: showcase:none-python-2.7
```

```
web:
  build: ../repo/Showcase/ShowcaseServer
  command: python manage.py runserver 0.0.0.0:80
  container_name: my_web
  depends_on:
    - db
    - orion
  environment:
    - PYTHONUSERBASE=../webapp/ShowcaseServer/
    - StartServers=2
    - MinSpareThreads=25
    - MaxSpareThreads=75
    - ThreadLimit=64
    - ThreadsPerChild=0
    - MaxRequestWorkers=150
    - MaxConnectionsPerChild=0
    - MinSpareServers=5
    - MaxSpareServers=10
  image: showcase:none-python-2.7
```

Figure F2: Default (top image) and amplified (second and third) Docker Compose configuration for CityGo web (Apache) service

These generated configurations are tweaking parameters related with Apache performance, with values constrained to suitable ranges, for those parameters related to performance (e.g. ThreadLimit, ThreadsPerChild)

Qualitative Evaluation and Recommendations

In this section Atos evaluation team in STAMP project conducts an qualitative and subjective evaluation of some STAMP tools (Table T7) by adopting a subset of the ISO 25010 quality model. Qualitative evaluation uses as metrics a 5-Likert scale [1], where 1 score corresponds to the lowest agreement with the fulfilment of the quality model (sub) characteristic and 5 score corresponds to the highest agreement.

3 Atos engineers have participated in this evaluation, and scores have been consensuated among them. At the time of writing, CAMP, DSpot and Descartes tools have been included in this evaluation. EvoCrash tool has not been included in current evaluation period (it will be in the next one).

Characteristic	Sub-characteristic	DSpot	Descartes	CAMP	Evocrash
Functional Suitability	Functional Completeness	4	4	3	N/A
	Functional Correctness	4	4	2	N/A
	Functional Appropriateness	4	4	3	N/A
Compatibility	Co-existence	N/A	N/A		N/A
Performance Efficiency	Time-behavior	2	3	4	N/A
Usability	Appropriateness recognisability	4	4	3	N/A
	Learnability	4	4	4	N/A
	Operability	2	3	4	N/A
Reliability	Maturity	3	4	2	N/A
Portability	Installability	4	4	5	N/A

Table T7: Quality Model based evaluation of STAMP tools in Atos UCs.¹⁴

In the following, we comment on the different evaluated quality model (sub)characteristics and provide feedback to developers. This feedback complements that one accompanying to the reported issues (within their discussion threads):

- **Functional Completeness:** both DSpot and Descartes are functionally perceived by the Atos development team as quite relevant and complete on what concerns the objectives of increasing the code coverage, by test amplification (DSpot) and the test quality, by detecting pseudo and partially tested methods. Both DSpot and Descartes are specialized in targeting unit tests, however, most of

¹⁴ In this table, CAMP, TECOR and Evocrash columns are empty as these tools are not evaluated in this period.

Atos test suites consists of system and integration tests. Exploring the benefits of DSpot and Descartes on these kind of tests could be also very helpful at industrial level. CAMP has also a high potential for Atos team. However, it is still unclear how CAMP can be configured to generate optimal deployment configurations for some relevant non-functional requirements such as performance, resilience, reliability, etc.

- *Functional Correctness*: initial experiments seem to prove that both Spot and Descartes results, namely: amplified tests and reports on pseudo/partial tested methods, respectively, are precise enough. Nonetheless, a detailed analysis of the results is pending to draw final conclusions on this regards. Besides, the analysis of the complete test suite could not be possible, due to the issues mentioned in above observations. Bilateral meetings between tool developers and industrial evaluators have been (and will be) hold to analyse the results and improve their correctness. CAMP results are preliminary, so the utility of CAMP for generating optimal configurations for addressing some non-functional requirements could not be determined. Atos is producing stress test that measure the performance of CityGo under different workloads and deployment configurations generated by CAMP. These tests will help to clarify whether or not CAMP configurations are appropriate to find CityGo deployments for optimal throughput.
- *Functional Appropriateness*: Both DSpot and Descartes tools are perceived by Atos team as quite promising for addressing our primary objective of improving our QA process and the test efficiency of our test suites, in order to prevent bugs in code (including regression ones). Preliminary analysis of Descartes results have already spotted some issues easy to solve that largely improve the test suite quality. Amplify tests produced by DSpot have increased the test coverage, despite DSpot could only be applied to a reduce subset of the entire test suite. CAMP is also perceived a quite promising for automating the testing of CityGo under different deployment configurations, but it is unclear if CAMP will be able to search for and generate optimal configurations, provided that non-functional models (i.e. for performance, etc) are given as input.
- *Time-behavior*: DSpot and Descartes runs are in general long time consuming. In the case of Descartes, current execution times are assumable (over an hour for the almost entire SUPERSEDE IF test suite), so Descartes execution can be integrated within DevOps CI/CD pipeline for important releases. DSpot execution takes longer, even for a single test suite, no consuming significant resources (i.e. CPU/Memory), but lasting even for hours, due to the high I/O intensive usage in Atos SUPERSEDE test suites. Further analysis of DSpot and Descartes performance is required. This challenge will be addressed by WP1 in next development period. Concerning CAMP, initial experiments are extremely fast, but further experiments under more realistic cases that explore a larger solution space are required to draw significative conclusions.
- *Appropriateness recognisability*: as aforementioned, Atos team acknowledge DSpot and Descartes tools as quite promising for addressing our primary objective of improving our QA process and the test efficiency of our test suites. Similarly for CAMP on the amplification of deployment configurations for optimal performance. On this regards, further experimentation will be required to extract the full potential of these tools when applied on our systems under test.
- *Learnability*: Both DSpot and Descartes offers good documentation for users. Current CAMP prototype is quite simple to use, but offers no configuration options at all:
 - [DSpot readme](#): as DSpot is being continuously developed, this documentation is sometimes outdated, leading to misuse by industrial evaluators, what caused some of the reported issues. Nonetheless, despite DSpot development has recently started (within the project), its documentation is rather quite complete and good. There are additional documentation in the [docs](#) folder of the GitHub repository, which is not in sync with the readme. Additionally, support provided by DSpot developers through the [GitHub tracker](#) is quite responsive for any user that post issues requesting for help.
 - [Descartes readme](#): as Descartes is a plugin for PITest, most of its documentation consist of [PITest documentation](#), which is quite complete. Nonetheless, specific [Descartes documentation](#) is quite complete, including references to further reading and an complete description of Descartes mission, usage, results, etc.
 - [CAMP Readme](#) mixes some new information with outdated one. It needs to be further

complete with instructions for configuration and execution beyond existing prototyping samples.

- **Operability:** As DSpot and Descartes are tools in continuous development, where not only the bulk modules but also their facades are evolving, we have faced issues (and we are still facing some of them) when running both tools in our experiments. Some of these issues are preventing us of executing both tools on our test suites. We are collaborating with tool developers to sort them out as soon as possible, so these problematic test cases can be targeted by the tools. For some issues, reported logs are not enough as they are not spotting the real causes of failures, giving us no clues to further investigate. On the other hand, several clients for DSpot and Descartes are being developed, including CMI, Maven and Gradle plugins, and also plugins for Eclipse IDE. We welcome all these client options to interface both DSpot and Descartes in different development processes, including traditional human-driven development in CMI and IDE, and also on the automated CI/CD pipeline adopting DevOps. The CAMP prototype we have experimented with offers a very simple CLI with no configuration options. Ongoing versions will be experimented in the next period to evaluate their configurability.
- **Maturity:** Despite the encountered issues that prevented us to applied these tools to the entire Atos test suites (for SUPERSEDE and CityGo UCs), we have got the impression that both DSpot and Descartes tools are significantly mature at this stage, because we understand these issues are platform specific. CAMP tool is in a less mature stage at this moment, with capacity to conduct some experiments in some use cases. Nonetheless, during the next evaluation period, jointly collaboration between tool developers and evaluators will be further developed.
- **Installability:** The installation of DSpot, Descartes and CAMP is rather simple from sources, and it is well documented. Other means of integration without the need of getting the sources, either using the Maven or Gradle plugin have not being evaluated yet.

9. Use Case : TellU

Introduction

Tellu provides cloud services for collecting and processing data, with a focus on IoT. We provide this to service providers and other partners in different domains, for integration in their solutions. Our services are built on our own software system: TelluCloud. Some key aspects of TelluCloud:

- Edges communicating with a multitude of devices, like GPS trackers, gateways and sensors.
- Aggregation of data in internal data model, with database storage.
- Logging of all received data and events.
- Processing of data by rule engine, with resulting actions.
- Integration with external systems through APIs.
- Easy management and configuration via the web.
- Customization, white label and re-branding.

Today we operate one main production instance of TelluCloud, used by a number of service providers (multi-tenant). The production instance is still based on an older 2.x branch of the system, while work has been underway on a 3.x branch for some time. Our **overall objective** in STAMP is to improve and automate testing and logging for TelluCloud 3.x, to ensure its correctness and robustness in a cost-effective manner so that it can be successfully put into production.

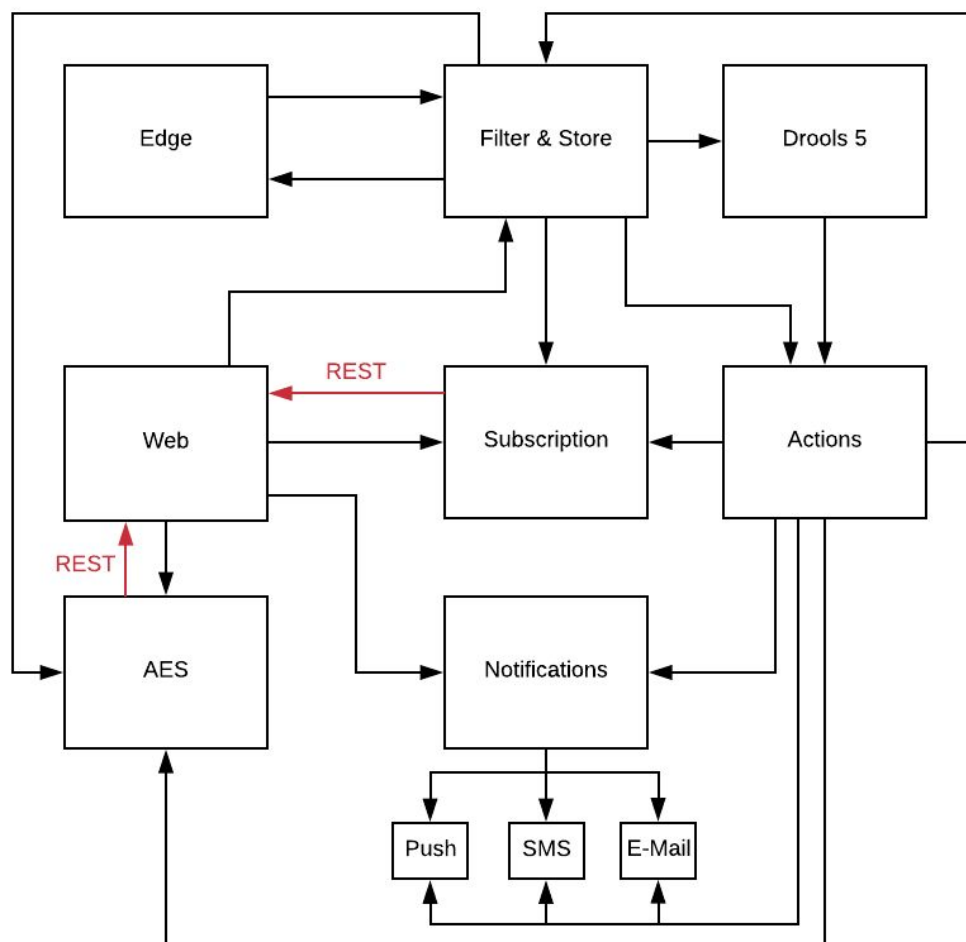
TelluCloud 3.x system

The TelluCloud 3.x branch represents a different system architecture and total refactoring of the system relative to 2.x. The main change was to switch to a micro-service architecture, splitting the main part of the system up into about a dozen micro-services. Flexibility and scalability have been the key requirements for this new system architecture. The new version is deployed in a cloud infrastructure, with persistent queues connecting the micro-services. Splitting up the system into smaller parts also makes the development and

maintenance of each part more manageable. However, it also introduces new challenges. The complexity of the system as a whole increases, especially with respect to configuration and deployment, as there are many more parts to deploy, including queues and other infrastructure. It becomes very important that each micro-service has well-defined APIs and protocols. In addition to unit testing, testing of each micro-service and of the system as a whole becomes paramount.

The TelluCloud 3.x branch has been going through much development and refactoring during the first part of the STAMP project. At the start of the project, we worked on a newly established 3.1 version. The source was organized as one multi-module Maven project with a large number of modules, hosted in a GIT repository at the hosting service Beanstalk (beanstalkapp.com). A new reorganization and refactoring was done for 3.2, splitting the source into many single-module Maven projects, each in its own GIT repository at GitHub. All TelluCloud source is still closed source, but we give STAMP researchers access to individual repositories when needed. TelluCloud 3.2 is still under development at the time of writing this report, and the current goal is to launch a production instance of 3.2 in June 2018.

The following figure shows the main micro-services in TelluCloud 3.2, with the main data flows. In a cloud deployment there can be several instances of each micro-service, and we need to handle load balancing and routing of messages to the correct instances. A distributed Service Registry keeps track of all instances.



STAMP use case focus

At the start of the project, we selected as our use case the TelluCloud 3.1 system of micro-services which were connected with queues, which at the time did not include web. This system, now expanded with more micro-services, continues to be the overall use case. The main flow starts with receiving observations from



sensors through various edge protocols in the Edge service, and continues with filtering, storing and rule engine processing. There are many possible outputs, handled by different services – different ways to send messages to users or external systems.

After the split into separate Maven projects and GIT repositories, we have selected three projects to focus on for unit test amplification (Work Package 1). These projects represent different levels in the source code hierarchy:

- **TelluLib:** In this library we keep code of a general nature, for use in any Tellu project. All TelluCloud modules have a dependency on this code. The code is well-suited to unit testing and had a good number of unit tests to begin with, running very quickly.
- **Core:** This library contains common TelluCloud service code, and is used by many of the micro-service projects. It contains the common domain model and several data access levels. It originated from the more monolithic main application of TelluCloud 2.x, with features gradually being refactored over into service-specific projects when possible, but is still the largest of all TelluCloud projects (in lines of code). It had unit tests coming from 2.x, but some of these are actually integration tests which takes some time to execute.
- **Filterstore:** This is one of the micro-services (Filter & Store in the figure). It is arguably the most important and most complex of the services, having a central position of receiving normalised data from edges, doing much of the initial processing and distributing the data to a number of other services. It is strongly dependent on Core. It has few, high-level tests.

For configurability test amplification (Work Package 2) and runtime test amplification (Work Package 3) we continue to consider the whole TelluCloud system of micro-services. In Work Package 2 we consider configuration of the whole system deployment, including cloud infrastructure. In Work Package 3 we look for appropriate logs from any micro-service, although we will prioritize logs from the three projects selected for unit test amplification.

Tellu objectives for STAMP

As described above, Tellu's overall objective is to improve and automate testing and logging for TelluCloud 3.x, to ensure its correctness and robustness in a cost-effective manner so that it can be successfully put into production. Tellu's objectives in STAMP are described based on the three forms of amplification in the project.

For unit testing, Tellu's objectives are to increase the number of JUnit tests for the TelluCloud source code, increasing test coverage, as well as improving the quality of the tests. The first part is easy to quantify with tools measuring test coverage. For test quality, our initial objective was to learn about state of the art practices and tools. This we have achieved in the project, learning about techniques such as mutation testing. With mutation testing we get a metric to help quantify test quality, so an objective is to compute this metric on our various projects and improve it. This should be integrated into the build chain.

For runtime test amplification, the initial objective was to get tools and methodologies to test the TelluCloud services at runtime, to check that running services are behaving according to specifications. More specifically, it is an objective to improve the quality of logging done by the system, and to get intelligent monitoring of the logs, to get automatic discovery and analysis of problems and inconsistencies.

Tellu is especially interested in configurability test amplification, with its aspects of managing diverse configurations and deploying a complex system in various configurations. TelluCloud is made to be deployed in different configurations. For deployment and test, it is important to be able to deploy both micro-services and full systems in a quick and easy manner. For production, it is important for Tellu to be able to move to different forms of cloud hosting depending on technological developments, pricing and customer needs. We foresee the need for at least two different production deployments. Some customers are best served with an instance running in Amazon's cloud infrastructure. But we are also heavily involved in e-health in Norway, and these customers want the data to be stored in Norway, requiring a more local solution.

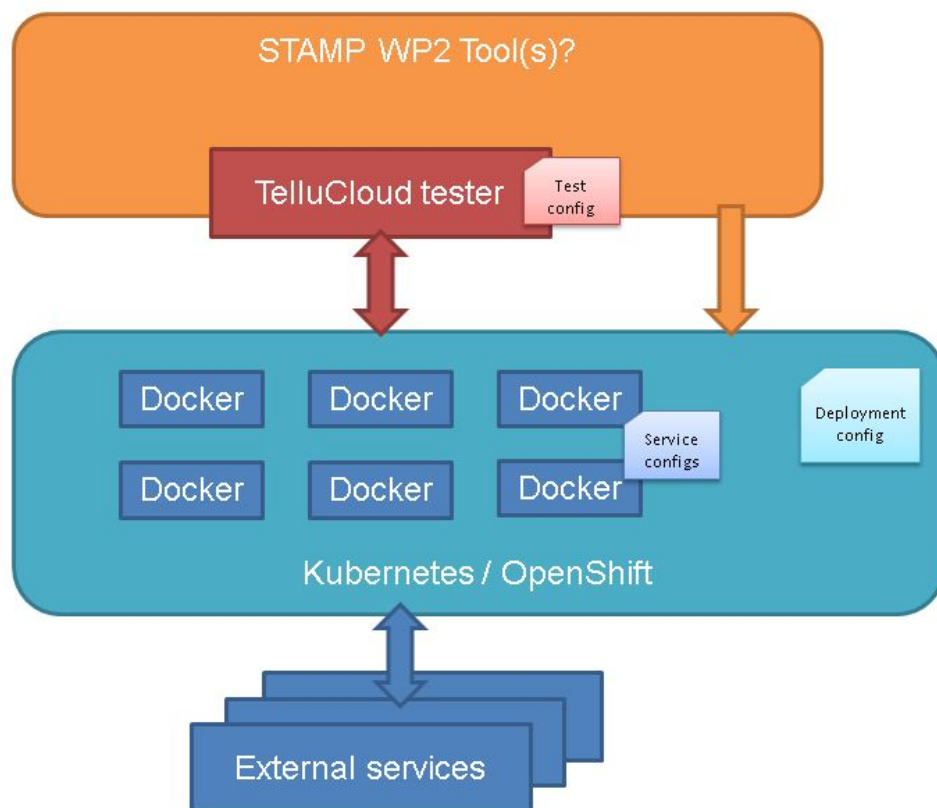
There are many factors which may vary:

- Configuration of each type of micro-service.

- Number of instances of each service.
- Data persistence configurations.
- Connectivity between micro-services, configuration of queues.
- Other infrastructure, such as service registry.
- Connections to external services.
- Type of deployment: single local machine for testing, server infrastructure, auto-scaling cloud deployment.

The first objective here is to orchestrate and automate deployment of the system, to facilitate efficient and automated testing. The next objective is to amplify configurations and test inputs, generating new variations of configurations based on existing ones. Finally, there are objectives to run different types of tests on the system:

- Functional tests: Verify correct service and system outputs based on inputs.
- Quantify performance and scalability, to find optimal configurations (performance vs price) and verify correct distribution and load balancing.
- Stress testing, ensuring that the response times and throughput latencies are within specified limits.



The above figure shows our vision for Work Package 2. Each micro-service is deployed as a Docker container. Tellu's interest in configuration testing is not just on the configuration of the individual services and Docker containers, but more on the level above – the composition and setup of the cloud deployment. We have been trying out cloud orchestration frameworks such as Kubernetes and OpenShift. With such frameworks, configuration for the cloud system level is specified in files. For running system tests, we need a tester that interacts with the system, giving it input and recording output. Our ultimate vision is to have a tool which can make deployments, varying their configuration, and run our system tests on the deployments, making sure the test configuration matches the system.



Finally, and common to all forms of testing amplification, is the objective to operationalize the tools and technologies. Tellu's STAMP objectives are tied to a goal to move towards DevOps. With the move to a micro-service architecture we want to be able to release new versions of separate services often. Strong automated testing of both the service and the system is a prerequisite.

Validation objectives for the period

Tellu's validation objectives for this assessment period can be categorized according to project work packages (forms of testing), and according to tools to validate.

For unit testing, we follow our overall objective of increasing the number of tests and test coverage. The project has a KPI of 40% reduction in non-covered code (KPI 1), and so we want to get started on the path to this goal with a 10% reduction. We also want to start seeing an increase in the number of lines of code tested per second (KPI 3). And we want to get a mutation score for our tests.

For configurability test amplification, the objectives are to be able to manage deployments with configuration files, and to be able to run system tests on the deployments, also defined by configuration files. Meeting these objectives will put us in a position to be able to do amplification, both on the deployments and the tests. For runtime test amplification, validation is tied to the EvoCrash tool, discussed next.

Two of the STAMP tools we have been testing in this period, DSpot and EvoCrash, have the objective of generating new unit tests. Our main validation objective with respect to these tools is therefore the generation of new tests. The last tool we are focusing on is Descartes, with the purpose of calculating mutation scores with extreme mutation. We therefore have the following objectives for mutation testing: First, to calculate mutation scores for our three selected TelluCloud projects. Second, the STAMP tool should produce mutation scores for the three projects. Descartes should produce scores which are useful, in the sense of giving meaningful insight into test quality, and do so significantly more efficiently than default PIT, completing in half the time or less.

Validation preparation

Here we summarize the work done on the TelluCloud use case for the purpose of evaluating STAMP.

Code development

TelluCloud 3.x has seen much development in this project period, including the reorganization of the source code into separate Maven projects and GIT repositories. This has meant refactoring of unit tests. In addition, STAMP work has contributed by writing new unit tests and improving the testability of the code.

Collecting metrics

A first set of metrics was collected in the first stage of the project. This was done on the 3.1 multi-module Maven project. Used tools included Jacoco for test coverage and SonarQube to aggregate statistics. As TelluCloud has been re-factored and split up for the current 3.2 version, direct comparison is not possible. We have chosen to focus the unit test validation on three Maven projects, as described above. Together they make up a significant part of the Tellucloud system, both in terms of lines of code and in their importance in the running system. They also represent a good mix of JUnit test cases.

We have also revised what tools to use for computing the metrics. For the first round, Jacoco was the best free alternative for computing test coverage. We set up a dedicated SonarQube server in AWS, needed to compute some of the other metrics as well as aggregate and visualize test coverage across Maven modules. Since then, Clover has become open source and a free version is now available. Clover appears to be a better and more modern tool for computing test coverage, and we have found we prefer it to Jacoco. It also provides the additional metrics we need, such as lines of code. We had been trying out SonarQube in STAMP, but since we no longer have the need for its aggregation from the multi-module project we stopped using it. So the 3.2 metrics have been collected with Clover in place of Jacoco and SonarQube, and this is what we hope to use for the remainder of the project. We have been learning about mutation testing in STAMP, and used Pitest to compute mutation scores as an additional metric. Here we have compared Pitest in its default configuration with Pitest running Descartes developed in STAMP.

The changes to the TelluCloud code means that we won't attempt any comparison between 3.1 and 3.2 metrics. We have also had to wait for some of the new projects to stabilize before computing metrics. We have computed WP1 metrics for the TelluLib project in December 2017 and April 2018, while we started in April with the two other modules. We think the code is now stable enough that we can continue to collect comparable metrics for the rest of the project for these three modules.

Regarding metrics to track KPIs for the other work packages, we have tried to compute what we could both at the start of the project and in April 2018, but not all are relevant for the Tellu use case.

TelluCloud testing tool

In order to run TelluCloud component and system tests, we needed utilities to produce input to the system, catch the corresponding outputs, and analyze the results. We have therefore been building up a toolbox of code to facilitate the system testing. While this was initially only for message-based input/output, it has been expanded to include other forms of interaction with the system, such as HTTP REST and more direct database interaction. This toolbox has gone through several iterations and has now become quite mature and generic. Parts of it could be useful to other projects, and could be made open source in the STAMP context. It is useful for both functional, performance and stress testing, or just to produce long-term load to a system. It can be used to test a single queue or micro-component, or a complete system. And it can be invoked in different ways:

- Run stand-alone for manual testing, producing a human-readable report based on a specific configuration.
- Programmatically from JUnit. We can then write tests with assertions on the aggregated results, allowing automation of testing.
- As a plug-in to some testing framework, such as JMeter, to take advantage of functionality found in such frameworks.

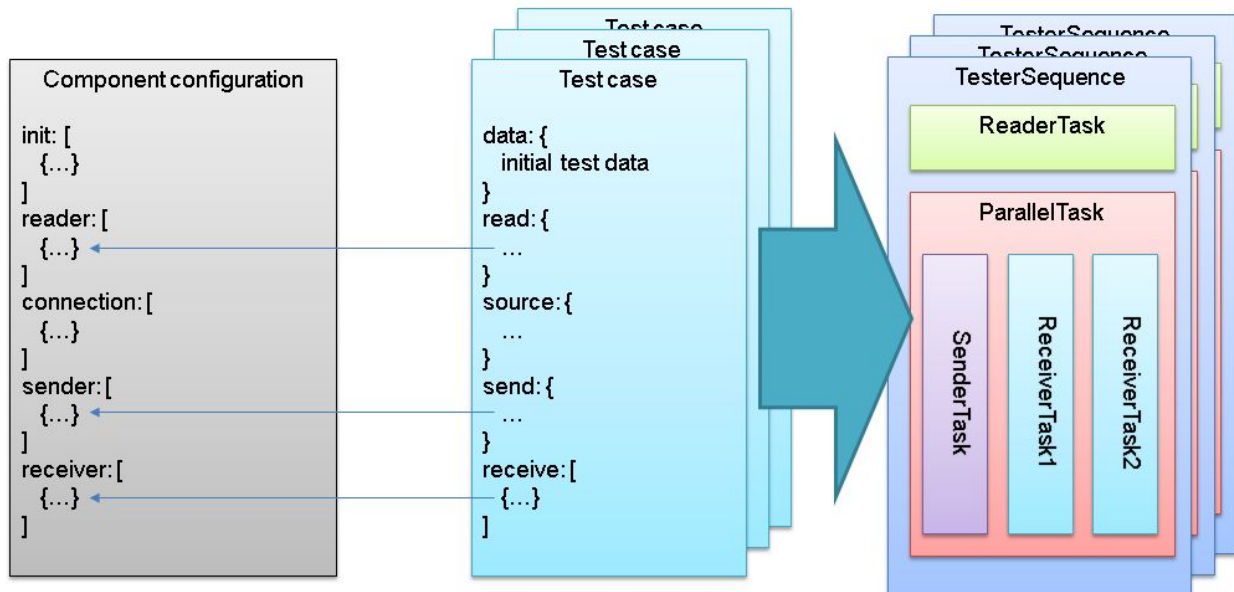
Architecturally, the code is organised in three layers:

- Tester components: These perform the actual interaction with the system, such as sending and receiving messages in different ways, as well as producing messages to send.
- Tester tasks: These handles running the components, with considerations such as timing, iterations, and parallelism, and produce statistics from the tests, with validation.
- TestManager: A top level to set up and run tasks and components based on configuration files.

On the component level, we have interfaces and abstract classes defining different types of test components, such as for providing messages, sending messages and receiving messages. We have a number of different implementations for these types, such as for sending via HTTP, RabbitMQ or Amazon queues. This allows running the same tests on different deployments simply by swapping out an implementation.

Tester tasks are used to automate the tests, and presents a uniform way of building a test. There is one task type for each component type, to run components of this type. Then there are composite tasks, so that tasks can be combined in a hierarchy of any complexity, to iterate over a sequence of tasks and run some tasks in parallel. Generic task functionality includes starting and stopping the task, and how the task logs time spent and number of transactions/sub-tasks run, to produce extensive statistics. Tasks can parse JSON data from components and also produces statistics in JSON form. Conditions on this data can be written in an XPath-like format, to do validation of test results. This makes it easy to validate the details of output from the system.

Our goal was to make it easy to define test cases and to modify and update test cases to run against different system deployments and configurations. We have therefore made it possible to specify test configuration and validation with JSON files. This is done with the top layer in our architecture, where a test manager can instantiate and configure components and tasks from the JSON configuration, and run the tests. The following figure shows how two types of configuration files produce a set of hierarchical tasks to run.



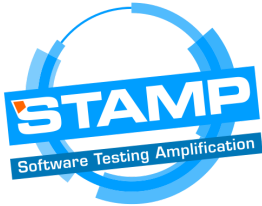
One file holds the component configuration, defining the components to be used in test cases. Each test case configures a hierarchy of tasks, using the components. These two aspects can then be configured independently of each other: we can change the component configuration to match a different deployment, running the same test cases, or we can add and change test cases, using the same components. The test case configuration includes conditions for validation. We automate the testing by invoking the test manager from Junit and making assertions on the validation done by the tasks once the test is completed. Now that all our test specification is done in a well-defined JSON format, it could also be possible to perform amplification of these tests.

Component and system tests

Development of the TelluCloud testing tool has been done in parallel with development of integration tests on component (micro-service) and system levels, with the tool developed to meet the need for such tests. These are the tests related to configurability test amplification. We have identified the need for the following levels of integration tests:

- **Component testing:** This is testing of each micro-service in isolation, and possibly other deployable components, such as database and queue. We want two types of test for each service. 1) Functional test, checking that the correct results are produced when messages are sent to the service. 2) Performance test, checking the number of messages processed per second.
- **Inter-component testing:** The TelluCloud system uses a mediator pattern to connect the micro-services, which includes handling of multiple instances of a service, with scaling and load balancing. These mechanisms need testing. We have identified three types of test. 1) Distribution correctness (that each message goes to the correct instance, in cases where this is significant). 2) Reorganization of shards: Shards are how we split up the messages to a service component type (f.ex. into a number of queues), and service component instances must correctly be reassigned shards each time the number of instances changes. 3) Testing of different configurations (instances, queues, etc.) to find optimal price/performance/load configurations.
- **System testing:** Here we test the whole system, on a real-world deployment comparable to production. Two types of test: 1) Functional test, checking that messages are correctly processed. 2) Stress/scaling tests.

We have implemented component tests for most micro-services, using the testing tool described above. These tests run both micro-service and test code together in a single Java VM and are not relying on external services or infrastructure. They are fully automated as integration tests run by Maven and Jenkins,



using JUnit. The test code first creates and starts the service, using a mock service registry. An in-memory database is used for services which need it, and the test code initializes this database with the needed entries. In-memory Java queues are used for input and output. The testing tool puts messages in these queues and subscribes to messages in the outgoing queues. Both functional and performance tests are implemented.

For the next level, the inter-component testing, we made some tests for TelluCloud 3.1 using the same mechanisms as for component tests. The test code would take down service instances and bring up new ones, to test sharding. These tests had limited value, since they used in-memory queues and had everything running in the same process. With version 3.2, it is no longer possible to run multiple TelluCloud micro-service instances in the same Java VM, so this testing approach is no longer possible. We now intend to test the inter-component aspects on a test deployment in the cloud.

For system testing we also made some single-VM tests for TelluCloud 3.1, running all services together with in-memory database and queues. This was useful to check that the interchange of messages works correctly. We also ran tests on a system deployed in Amazon's infrastructure (AWS), and this has been the focus since TelluCloud 3.2. Here we post data to the edges over HTTP and TCP, with the tester tool mimicking sensor devices. Output channels such as WebSocket and SMS are monitored. We have now automated these tests, running the tester tool with JUnit from our build server every night. We have also developed API tests for TelluCloud's REST API. However, these tests rely on the AWS deployment, and we do so far not have automatic ways to make deployments or vary their configuration.

Deployment configuration

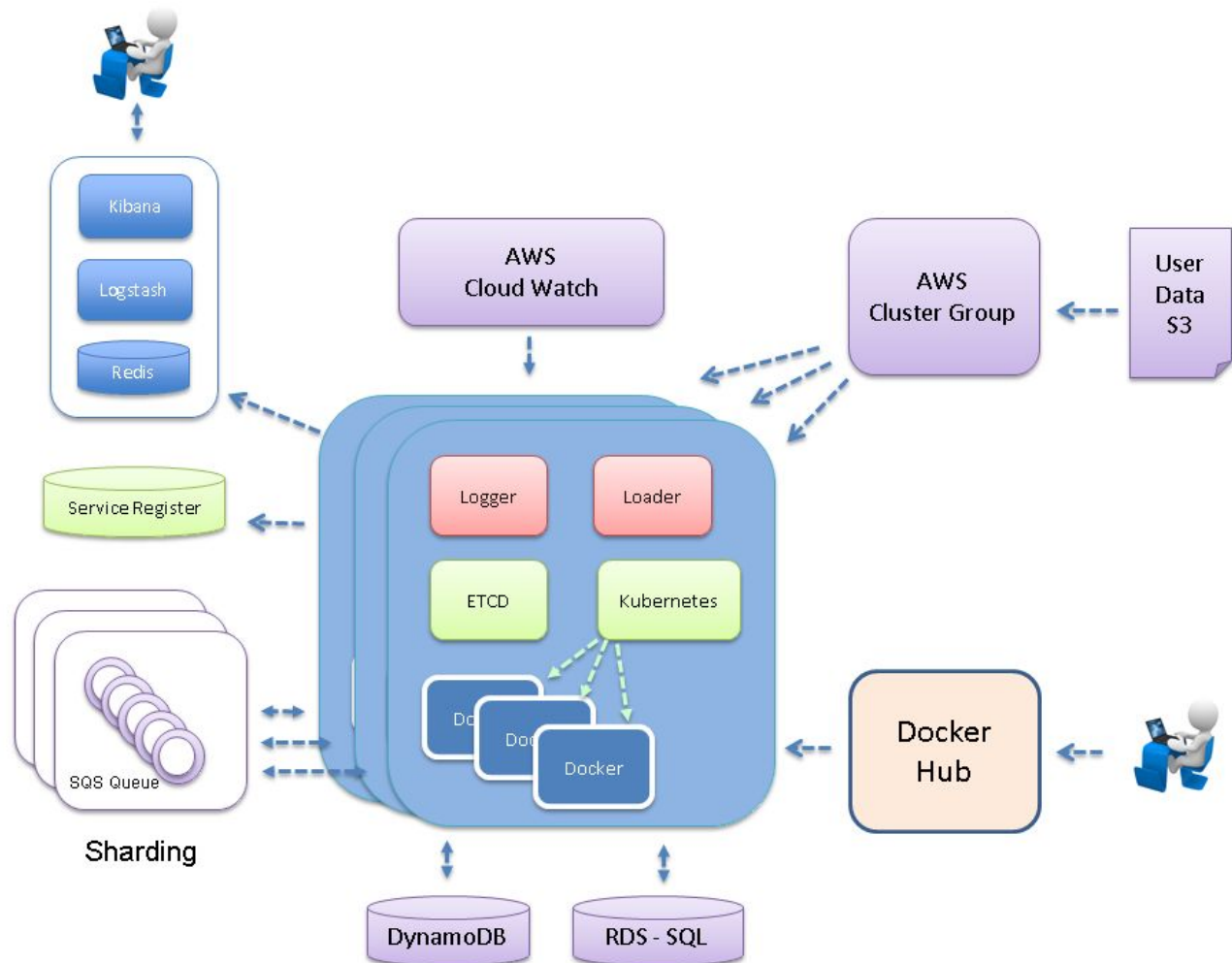
As we have already discussed, a TelluCloud deployment involves different forms of configuration, on different levels. The micro-service architecture and large degree of flexibility means there is a large configuration space. One of Tellu's tasks has been to map the configuration space, getting a complete overview over all internal and external parameters, how they are specified and which values we want to try for these. We have been working on how configuration is done, developing this aspect of the TelluCloud system. There are also significant relationships between the different forms of configuration. For instance, services need to be configured correctly to connect to other services, and this needs to match the infrastructure and system configuration. When we bring in system tests with the TelluCloud tester tool, the tests also need to be correctly configured to interact with the deployment to test.

A significant part of Tellu's work has been on how to handle system deployment. We needed to do a lot of learning and experimentation, as this was a new area for us. For formalizing and automating the testing as much as possible, we need a way to specify and automate deployment. The approach was to use an orchestration framework, which handles much of the work based on configuration files. We wanted an approach which can be used for everything from a local deployment on a test machine to a full cloud deployment. For cloud deployments, we need to support most hosting providers, and have been using both Amazon and a Norwegian provider.

We settled on Kubernetes, and developed configurations for use with it. We then switched to OpenShift, which adds another layer, building on Kubernetes. But this adds requirements on the infrastructure, and so, after testing both, we ended up back with Kubernetes. Kubernetes appears to be a good match for our case and for configuration testing. It is increasing in popularity, and some cloud providers have built-in support for it. Otherwise it can be set up to run anywhere. It can also be used to deploy locally, with the tool Minikube making it easy to set up a single-node cluster in a virtual machine on any PC. Most importantly, Kubernetes allows specifying deployments in configuration files. This means that much of what used to be external configuration in selecting infrastructure components and resources can be explicitly stated in configuration files. This greatly facilitates configuration testing and automation of deployment.

The figure below illustrates a deployment architecture, in this case on AWS. We deploy the service components in Docker containers using Docker Hub. In this particular configuration we have database implementations provided by Amazon (bottom) and use Amazon's SQS queues to implement messaging between services. And we see a logging stack, with Redis, Logstash and Kibana (top left). All of this can be

changed. Databases and queues can be handled with Docker, making this part of the configuration handled by Kubernetes.



Validation results

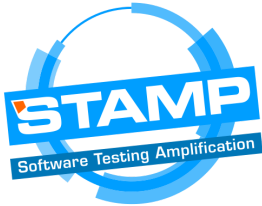
Here are Tellu's results for the STAMP tools we have tested so far.

Descartes

Running Descartes on the main module of the TelluCloud 3.1 source, the process eventually stopped with no error message, and had to be aborted. Running PIT as normal produced a similar result.

For TelluCloud 3.2, Descartes is used on the three selected projects. It runs successfully on all three, and it was easy to set up following the instructions on the GitHub page. We have been running PIT in its default configuration (without Descartes) for comparison. See table for results.

	Descartes mutation score	Descartes runtime	Pitest mutation score	Pitest runtime
TelluLib 2017	35%	46 sec	29%	113 sec



TelluLib 2018	46%	78 sec	44%	197 sec
Core 2018	17%	21 min	Did not work	
Filterstore 2018	52%	7:30 min	40%	28 min

Note that for the Core project, default PIT did not even work, spitting out a 0% score shortly after starting, while Descartes was able to run and produce a relevant score. For the rest, we see that Descartes was faster by a factor of 2-4. The mutation score it produces is consistently higher than what we get with default PIT. It is less accurate, but not drastically so, and it seems consistent across versions of the code and with respect to other projects. We believe the mutation score produced by Descartes is a useful indicator of test quality, and will continue to use it instead of default PIT for TelluCloud mutation testing due to the significantly lower processing times.

DSPot

Getting DSPot to run correctly was troublesome, and only possible with help from the developer. We have so far been testing it on the TelluLib project. The last test so far was at the beginning of 2018. We tried all the different mutators available at that time. Running two iterations took from half an hour to over two hours, depending on the mutator. Most tests would end with zero new tests. Only one test gave a result to look at. This was with mutator "StatementAdd". Two iterations took 55 minutes. It reported two new tests killing one more mutant. On closer inspection, the generated tests seem to do the same as one of the original tests. The details have been posted on GitHub¹⁵.

The result is that DSPot has not produced useful results for TelluCloud so far. We have since been focusing on the other tools but will return to DSPot to test a new version.

EvoCrash

We have done two types of initial testing of EvoCrash. The first is getting the tool itself to run, using provided cases. This was initially hard to do, as the "Getting started" instruction did not include instructions for Windows. In cooperation with the developer we got it running. The test case XWIKI_13916_Test (one of the "Getting started" cases) ran but did not complete. It probably ran out of resources. We were able to run one case, XWIKI_13031_Test, to completion, generating a test which replicated the frames of the stack trace.

With help from the developer, we tried EvoCrash on two TelluCloud cases. These were crashes with stack trace in two different services. In one case, EvoCrash could easily generate a test from any of the first three frames of the trace. Beyond this it was difficult to generate a test, and reasons for this was identified by the developer. The other case was much more problematic. The method where the crash happened was too complex, with the crash happening after several hundred lines of code and many conditional statements. EvoCrash is not able to replicate this run, and it is not feasible to do so when it relies on satisfying a high number of conditions within one method.

The result of this initial evaluation is that EvoCrash shows promise, and that much work remains, both for the tool to be further developed and for Tellu to validate the usefulness of the generated tests and its role in the DevOps toolchain.

Quality Model

Characteristic	Sub-characteristic	DSPot	Descartes	CAMP	Evocrash
Functional Suitability	Functional Completeness	3	5		2

¹⁵ <https://github.com/STAMP-project/dspot/issues/291>

	Functional Correctness	1	3		1
	Functional Appropriateness	0	4		1
Compatibility	Co-existence	5	5		5
Performance Efficiency	Time-behavior	1	4		3
Usability	Appropriateness recognisability	2	4		2
	Learnability	1	3		1
	Operability	1	4		1
Reliability	Maturity	2	4		1
Portability	Installability	1	5		2

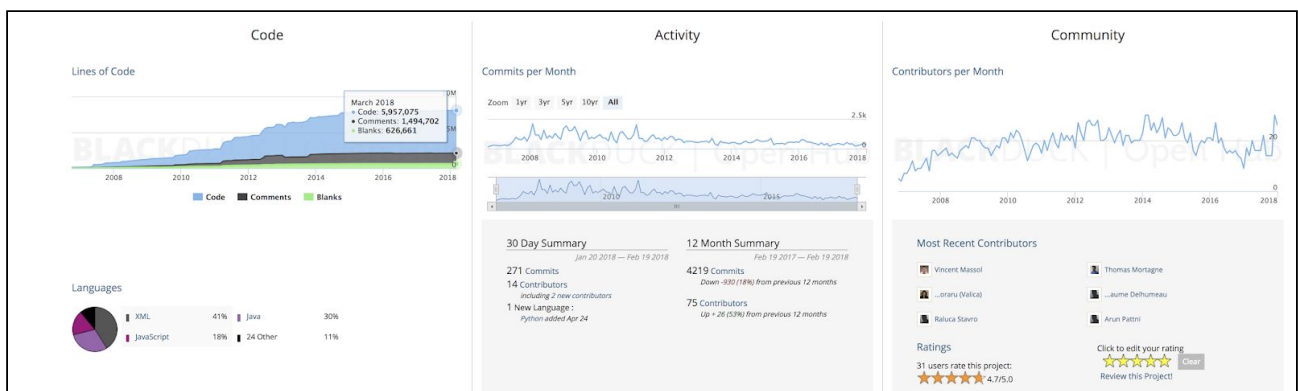
10. Use Case : XWiki

Introduction

[XWiki](#) is an advanced Enterprise open source wiki tool written in Java but it's also a [generic web-development platform](#) that can be used to develop any kind of web applications, especially if they relate to content. Its [features are numerous](#) and this makes it a large and complex platform.

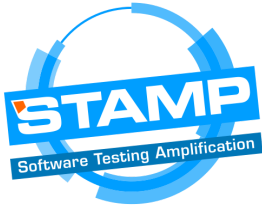
XWiki currently boasts more than [700 extensions](#) providing additional features (Blog, Meeting Manager, LDAP integration, etc).

The whole codebase size varies between 6M lines of code as reported by [OpenHub](#) and for the whole platform with lots of extensions included, down to between 300K ([SonarQube](#)) or 650K ([Clover](#)) for the Core product that is called XWiki Standard.



For the STAMP project we've decided to focus on XWiki Standard. The code for XWiki Standard is spread across 3 repositories in [GitHub](#):

- `xwiki-commons`: common modules that can be used outside of XWiki



- `xwiki-rendering`: the XWiki Rendering engine (i.e. libraries used to convert an input in a given syntax into another target syntax, e.g. from XWiki Markup to HTML)
- `xwiki-platform`: all the wiki features

This means that we're applying the tools developed by STAMP on those 3 code bases and that the KPI/metrics are measured on them too. We could have taken a smaller subset but we decided to take the whole product code base in order to have both a significant code base but even more importantly to be using real production code from various domains: Database access, Web UI with Javascript/CSS/HTML, etc. We wanted to make sure that the STAMP-developed tools could work for those various use cases/needs.

The following use cases are of interest to the XWiki project:

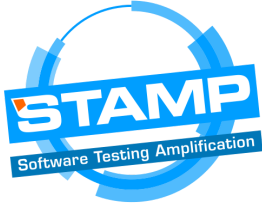
- **Improving Flaky Test handling**: Several times per week, [XWiki's Continuous Integration server](#) (CI) sends false positives to the XWiki open source project's notifications mailing list. This list is used by XWiki developers working on the project. When they receive such emails, they usually stop what they're doing to analyze the problem, spending several minutes or hours to finally discover that the problem is actually either a build environment problem or a flickering test. This is wasting their time and preventing them from progressing faster on their task at hand. In addition, it decreases the value given to the CI, and, over time, developers stop caring about build failures coming from the CI till the day of the release and then all integration problems need to be fixed, this delaying the release a lot, and negating the exact purpose of a CI tool which is to discover integration problems as they happen.
- **Improving test coverage and test accuracy**: XWiki already has a relatively high level of test coverage (around 70% at the start of the STAMP project). However, the community raises bugs every day, showing that we still need more tests. Test coverage is not an absolute measure of the quality and getting to 100% is not the goal but increasing it to around 80% would help a lot. Especially in parts of the code where this coverage is low currently. Now test coverage doesn't guarantee the quality of the test (a test with no asserts will generate coverage but won't prove much). Thus it's also important to evaluate the quality of the tests. This is where mutation testing can help out. In addition it would be awesome to have tests automatically generated and that would increase automatically the test coverage/accuracy.
- **Configuration testing**: XWiki is web application deployed as a WAR which can be installed on any Servlet Container and running on any Database. It can also run in all browsers. Thus it's very important that the XWiki tests are executed in various environments to prove that the [subset of supported environments](#) work, and that no regression are brought to any of those environments when new code is added. In addition and to make XWiki's distribution simpler to set up, we want to package those environment setups as Docker images.
- **Reduce time to debug runtime issues**: When XWiki users have a problem in production, they report it through the [XWiki issue tracker](#). They are asked to include the stack trace in the issue report. It would be nice and interesting to make it simpler to reproduce the issue by having some tool generate automatically the test, that, when executed, leads to this stack trace. This is the ambition of EvoCrash and something the XWiki project is keen to test.

During the period, we developed some tools to help us achieve some of the listed use cases (more details below) and we tested the following other STAMP tools: PITest/Descartes, DSpot, and EvoCrash. We also looked at CAMP but spent less time on it since we were more interested in testing specific well-defined configurations than doing mutation/amplification of configurations.

Validation objectives for the period

Main objective for this period were:

- Do a first round of testing of the following STAMP tools: Descartes, DSpot & EvoCrash



- Do this first testing round on `xwiki-commons` and `xwiki-rendering`, leaving `xwiki-platform` for later when the tools are more mature (`xwiki-platform` contains more complex tests, including automated functional UI tests using Selenium/WebDriver).
- Define a strategy useful for the XWiki project about how to use these tools.
- Develop some scripts and simple tools required (e.g. to measure the full Test Coverage of XWiki Standard, more information below)
- Do a baseline measure of KPIs at the beginning of the project and another one at the end of this period so that we can measure progress.
- Provide use cases and early feedback for Descartes, DSpot and EvoCrash

Validation work

Validation preparation

The following tools were developed during this first period:

- The XWiki CI setup was converted from “traditional” Jenkins jobs to Pipeline scripts, with a generic Pipeline Library developed for building Maven project.
- A tool was developed to handle Flaky Tests.
- A Jenkins Pipeline script was developed to fully automate the computation of the Test Percentage Coverage (TPC), using Clover, for the whole of XWiki Standard.
- A Groovy script was developed to harmonize Clover Test results and be able to compare progression.
- Docker images for XWiki were developed for various environments, as a prerequisite for being able to automate tests on various setups.
- Developed JUnit5 extensions so that XWiki tests can be moved from JUnit4 to JUnit5, as a preparation for executing/validating the STAMP tools on JUnit5 (Descartes, DSpot).

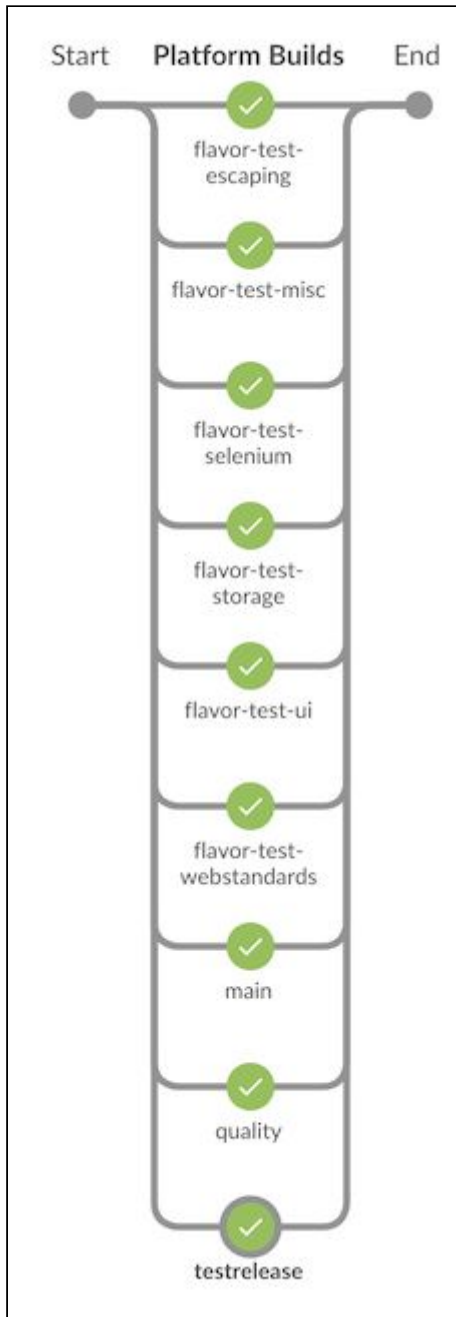
Jenkins Pipeline Build

XWiki was using standard Jenkins job and it had to [move to Pipeline jobs](#) as a prerequisites for implementing the various strategies for applying the STAMP tools on XWiki.

A [generic pipeline library](#) was developed which does the following at a high level:

- Choose version of Java by inferring it by reading the Maven POM file
- Maven build, with a timeout to stop stuck jobs
- Allow XVNC to be executed (needed for functional UI tests)
- Send failure notification emails
- Handle flaky tests (more below)
- [Attach screenshots](#) for failing UI tests executed by Selenium

This allows to have [simple Jenkinsfile](#) to execute various builds in parallel to execute the functional tests, run the quality checks, etc.

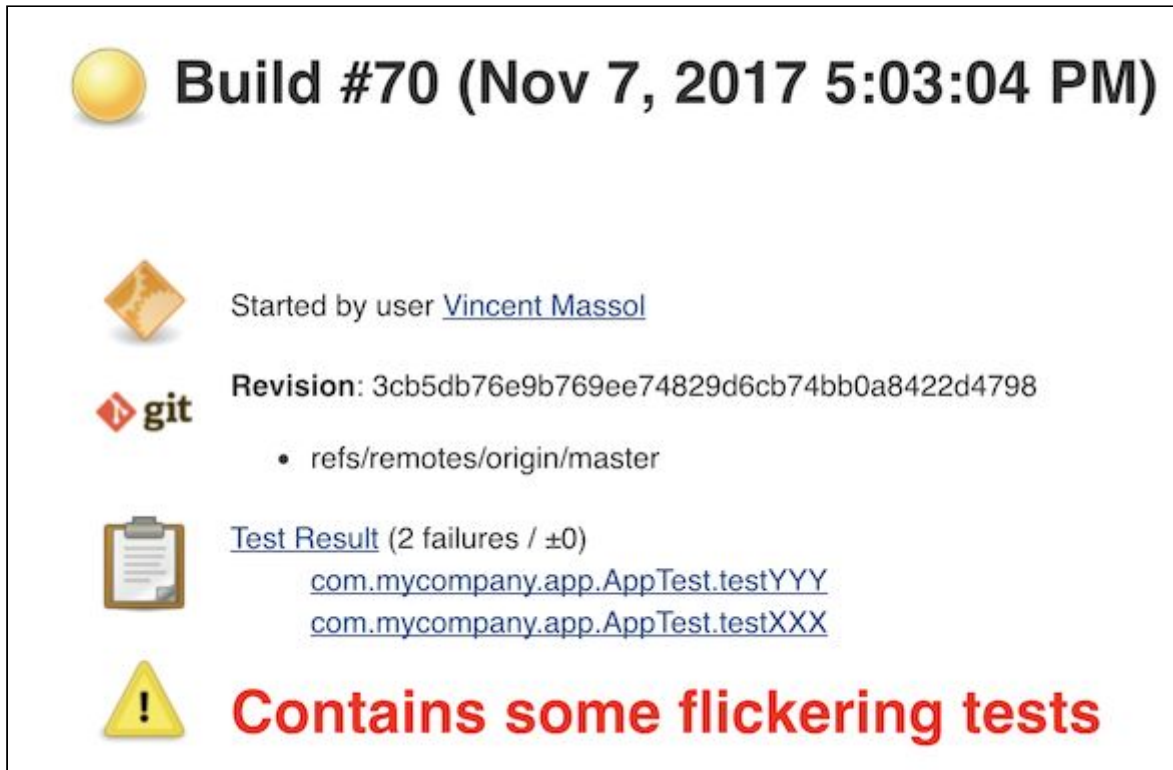


Flaky Test Handling

[A tool was developed to handle Flaky Tests](#) and prevent false positive emails. It works as follows:

- A custom field was added in the XWiki JIRA instance
- When a Flaky test is discovered, a JIRA issue is reported and the custom field is set with the name of the failing class/test.
- The Jenkins Pipeline library developed has been modified:
 - It queries the XWiki JIRA instance to list all registered Flaky Tests.
 - When the build fails because of a test, the library checks if the failing tests are flaky tests or not.
 - If all failing tests are flaky then it doesn't send a build failure notifications email.

- It modifies the Jenkins Job page to report the Test as being a flicker and the job as containing flickering tests



Build #70 (Nov 7, 2017 5:03:04 PM)

Started by user [Vincent Massol](#)

Revision: 3cb5db76e9b769ee74829d6cb74bb0a8422d4798

- refs/remotes/origin/master

Test Result (2 failures / ±0)

[com.mycompany.app.AppTest.testYYY](#)

[com.mycompany.app.AppTest.testXXX](#)

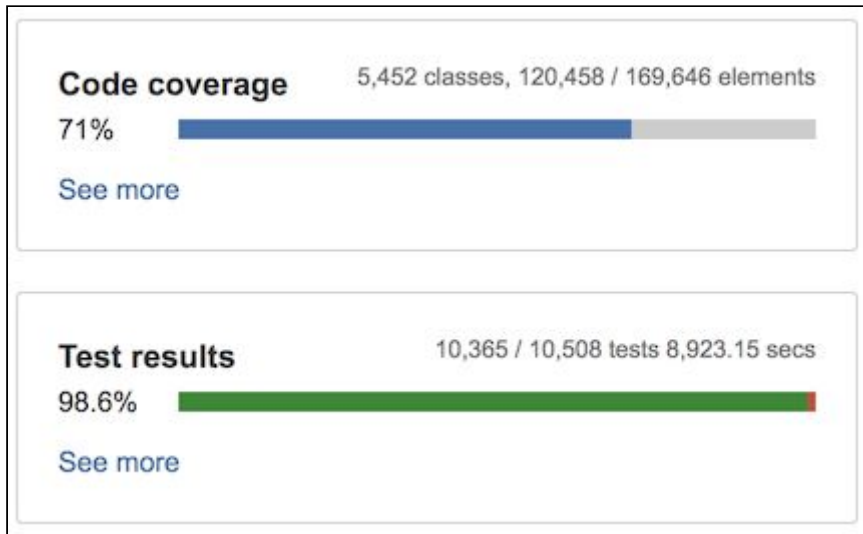
Contains some flickering tests

Global Test Coverage

A Jenkins Pipeline script was developed to fully automate the computation of the Test Percentage Coverage (TPC), using Clover, for the whole of XWiki Standard.

It's easy to use Jacoco or generate the test coverage for a single reactor build of Maven. However it's a lot more complex when your build is made of several builds (i.e. several Maven reactor executions). For this XWiki developed a [Jenkins Pipeline script using Clover](#). Since it takes a long time to execute (5 hours+), this script is executed once per month. The results are then [published](#).

For example here's the [report for April 2018](#) for the whole of XWiki Standard (xwiki-commons, xwiki-rendering and xwiki-platform).

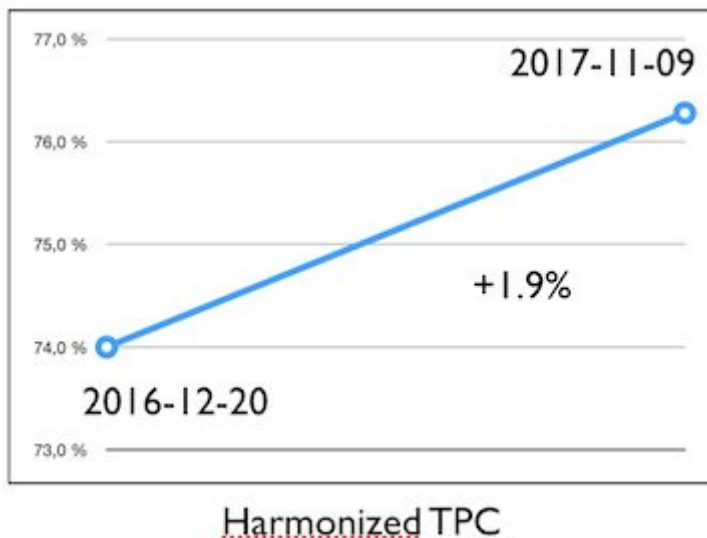


Comparing Clover Reports

In short, when you generate several reports with Clover and look at the global TPC figure, you're getting some weird results. This is caused by some discrepancies in how Clover separates main code from test code and thus on what it computes the coverage. It also depends if you've configured Clover to instrument test source and if you're measuring coverage for test code (like test framework code) or not, and how these config evolved over time.

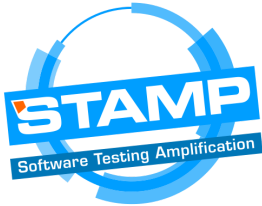
In order to be able to compare Clover reports, we've thus [developed a Groovy script](#) that gets the raw XML data and compares 2 Clover results.

For example from 2016-12-20 to 2017-11-09 we can see that XWiki increased its coverage by close to 2%:



XWiki Docker Images

In order to be able to test XWiki on multiple environment, a prerequisite was to first [develop some XWiki Docker images](#). We did this for the following combinations:



- XWiki 9.x/10.x on Tomcat, using a MySQL database
- XWiki 9.x/10.x on Tomcat, using a PostgreSQL database

These images were released as part of the XWiki project and have been made available as [official Docker images on DockerHub](#) (totalling more than a million pulls at the moment of writing!).



The following work has been started but is not finished yet:

- Include the generation of these images [as part of the XWiki build](#).
- Contribute an image for [XWiki on Oracle Database](#).

JUnit5 Conversion

The XWiki project has also started transitioning from JUnit4 to JUnit5, as way to prepare for testing the various STAMP tools on JUnit5. JUnit5 is the new standard and projects are starting to move to it. It's thus important that the STAMP tools work on the latest version of JUnit.

Performing this migration is not a small feat though since the XWiki project had a lot of custom extensions for JUnit4 (mostly in the form of `@Rules`) and since [Rules don't exist anymore](#) in JUnit5, there was the [need to rewrite them all](#).

Validation result

PITest/Descartes

Descartes [was tested on xwiki-commons and xwiki-rendering](#). Reports were generated and some tests were fixed as a result of executing Descartes ([example 1](#) and [example 2](#)). Every time commits were done, the test coverage was raised as a consequence.

We [learnt](#) the following:

- It's slow to execute. For example here are times found when building xwiki-commons with and without Descartes:
 - Without Pitest/Descartes 5:10 minutes
 - With PITest/Descartes: 37:16 minutes. That's over 32 minutes more, or 740% of increased time.
- It takes a lot of time to analyse the generated reports and the ratio time to analyse vs added value of fixing the tests is very often not worth it.
- However it's interesting to classify the results in 3 categories:
 - **strong pseudo-tested methods**: no matter the return values of a method, the tests still passes. This is the worst offender since it means the tests really needs to be improved. This was the case in the example above.
 - **weak pseudo-tested methods**: the tests passes with at least one modified value. Not as bad as strong pseudo-tested but you may want still want to check it out.
 - **fully tested methods**: the tests fail for all mutations and thus can be considered rock-solid!
- Thus the Descartes report should be modified to highlight/make it clear what are the strong pseudo-tested methods since they are the interesting code to analyse.



- We found a big limitation: PIT doesn't support being executed on a multi-module project. This means that right now you need to compute the full classpath for all modules and run all sources and tests as if it was a single module. This causes problems for all tests that depend on the filesystem and expect a given directory structure. It's also tedious and a error-prone problem since the classpath order can have side effects. As a consequence the [pitmp Maven plugin](#) was developed.

On the XWiki project we're mostly interested by what we call "Active Quality", i.e. quality checks that make the build fail. By contrast "Passive Quality" would mean dashboard or reports generated that devs would need to read/analyze and voluntarily decide to fix things based out if it. In practice our belief is that Passive Quality doesn't work and that devs are too busy to do this.

Thus we wanted to find a way to use Descartes and make it an "Active Quality" item for XWiki. The strategy [we decided](#) to put in place is [the following](#):

- Add a profile in the Maven build to check the mutation score of each module and to fail the build if the computed score is below a threshold defined in the `pom.xml`.
- Compute the current mutation score for all modules and set that score as the base threshold to use for now.
- Since executing Descartes takes a long time, create a CI job on Jenkins and don't execute this job at each commit. Instead run it once per week.
- Send a notification email when one module's mutation score goes below the threshold, thus signalling to the developers that the quality of the tests of that module has decreased. Since the build is failing, XWiki won't be able to be released before the violation is fixed.

Here's a high level list of limitations noticed when using PIT/Dcartes:

- Slow to execute
- Doesn't currently work with JUnit5
- Failing on some modules
- We also need to [better understand how the mutation score evolves](#) when code evolves (i.e. when new code is added or removed and when new tests are added and removed).
- We need more feedback on the long term about the strategy put in place on the XWiki project. At the moment of writing, we've not had a module failing yet because of a quality drop in tests. When this happens, we'll need to check with the developer and verify that he agrees that he's lowered the quality. This will be the real test to verify that the strategy works (or not)...

In total [11 issues were reported over the period](#) in GitHub issues (and numerous ones discussed and reported verbally during work sessions).

DSpot

DSpot was tested mostly on xwiki-commons. Several testing sessions were done since early tests didn't yield any results, i.e. no new tests were generated.

During the last testing session we were able to generate [one new test after running DSpot on 6 xwiki-commons modules](#).

Results are not great yet but DSpot holds an very interesting promise: that of automatically generating new tests. While PIT/Dcartes will only tell you about the quality of your tests, DSpot goes one step further and tries to help you by generating new tests that increase the quality and test coverage for your tests. That's a very interesting aspect since it doesn't require human intervention.

Right now and after testing DSpot on the XWiki project we don't feel it's able to generate enough new tests to be truly useful but as it progresses and is able to do that better, it may become good enough at some point to be truly useful.

The strategy we'd like to put in place to have it in the "Active Quality" area is the following:

- Create a Jenkins pipeline job which executes DSpot on the XWiki code
- Since it's time consuming, run it only every month or so
- Have that pipeline automatically commit the generated tests to XWiki's GitHub repositories, in a different test tree (e.g. `src/test-dspot/`)
- Modify XWiki's Maven build to use the [Build Helper Maven plugin](#) to add a new test source tree so that the build runs on both the manually-written tests and the ones generated by DSpot

Implementing this strategy will be done in the next period, using the [new DSpot Maven plugin](#) when it's ready.

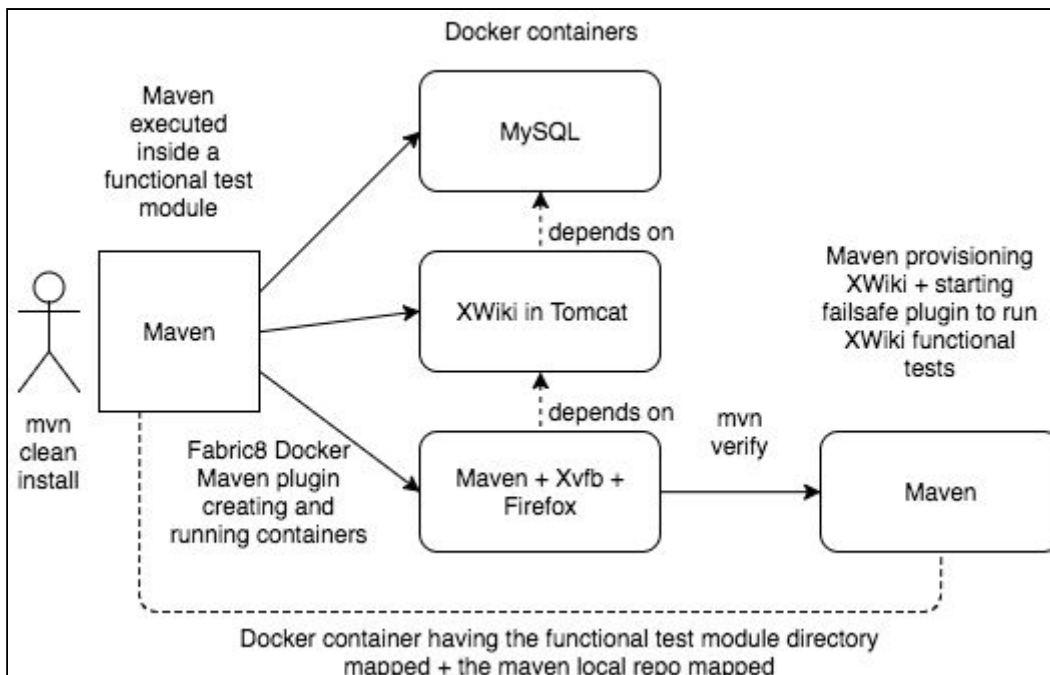
Note that we also tested using the Clover selector instead of the default PIT/Dcartes selector. This means changing the strategy that DSpot uses to keep new tests. By default DSpot keeps tests that kill more/different mutants than the original test. Using Clover, DSpot keeps tests that increase the test coverage. However at this time of writing our test session have not yielded any significant result since it's [not stable enough](#). We hope to be able to test this again in the second period when it's more stable.

Configuration Testing

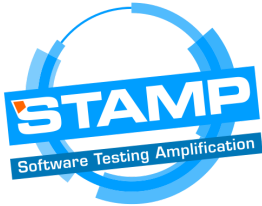
XWiki started by developing various Docker images that contain XWiki setup in various databases and various versions of XWiki (see above).

The next step was to be able to automatically execute XWiki's existing functional tests inside those various Docker images.

Initially we [imagined running Docker inside a Jenkins pipeline](#). However we dropped the idea since one goal we had in mind was to be able to execute the functional tests on the developer's machine (and thus not only on the CI). This is why in our second architecture below, we instead focused on integrating Docker execution as part of the XWiki build.



We modified [XWiki's build](#) in the following aspects:



- We added some modules under xwiki-distribution to generate the XWiki Docker images as part of the build (vs previously generating them in an adhoc manner and separately from XWiki's build).
- We also added a new Docker image that will have Maven installed + Xvfb + all browsers we'd like to use to execute the functional UI tests (e.g. Chrome, Firefox, etc).
- We started exploring the usage of the [Fabric8 Maven Docker plugin](#) to execute Docker

The idea is that the Maven build will use the Fabric8 Docker plugin to launch 3 containers:

- One for the database to test
- Another one containing the Servlet Container with the XWiki WAR deployed in it
- A third one that contains the browsers and that will be used to start Maven so that it executes the functional tests

A [POC was implemented](#) and is close to working. It has one important issue that needs to be fixed: right now the execution is very slow and needs to be tuned. We believe the issue is that the execution of Maven which spawns Docker which spawns Maven which executes the functional tests is taking too much memory by default (by default Docker containers will take a lot of memory if not constrained). Thus we need to finalize tuning the memory to make it work nicely and transform the POC into a production-ready usage. This is meant to happen during the second period.

EvoCrash

Several production stack traces were selected by [searching for them in the XWiki Issue tracker](#) (JIRA) and using the following query: `description ~ '.java:' AND description ~ 'Exception' AND category = "XWiki Standard Flavor" AND createdAt >= '-365d' and resolution not in ("Cannot Reproduce", Duplicate, Inactive, Incomplete, Invalid, "Won't Fix") ORDER BY createdAt DESC`.

Namely, [the following issues were selected](#) and Evocrash succeeded to generate tests for them:

- XRENDERING-422
- XWIKI-13616
- XWIKI-14475
- XWIKI-14612
- XWIKI-13031
- XWIKI-13196
- XWIKI-13916

Here are some detailed analysis that were done on them:

- [XRENDERING-422](#): Even though Evocrash reproduced a similar stack trace, it's not showing the real problem. The generated test simply shows that if you call an `endXXX()` event without calling a `startXXX()` one then you'll end up with some `NoSuchElementException` caused by the `pop()` without a `push()` being done. So the test generated by EvoCrash is not useful here.
- [XWIKI-14475](#): The real problem is actually not in `DefaultWikiTemplateManager` but in `DefaultWikiPropertyGroupManager` which gets injected `Map<String, WikiPropertyGroupProvider> propertyGroupProviders` once (when the singleton is loaded the first time). The issue is when `xwiki-platform-wiki-*` modules are installed as extensions (when provisioning a new subwiki for example). It's not possible for EvoCrash to know about this since `DefaultWikiPropertyGroupManager` doesn't appear in the stack trace anywhere... Thus, the test generated by EvoCrash is not that useful in this case since it's easier to understand the code than the generated test. However, what's nice with test generated by EvoCrash is that it makes it easier to understand some conditions that can lead to a problem. The developer still needs to figure out the real reason of the problem and write a proper test but it can help.

So far it seems that the tests generated by Evocrash are not that useful. I believe they could be useful in some cases:

- For example a `NullPointerException` happening on a line where several object can be null and thus where a visual inspection wouldn't reveal the issue.
- For developers new to the code since it could help them begin to understand the problem, even though they would need to work their way to find the root cause. For senior developers knowing well the code base, it's less useful.

However more analysis are required for the other results in order to draw any significant conclusions. That'll be the topic of the next period.

Quality Model

For each sub-characteristic and for each tool you used, provide your subjective degree of satisfaction (find definition for these characteristic in the appendix or https://edisciplinas.usp.br/pluginfile.php/294901/mod_resource/content/1/ISO%2025010%20-%20Quality%20Model.pdf) please use scale (0 : worst - 5 : best, ? for N/A)

Don't need to fill unused tool/column

Characteristic	Sub-characteristic	DSpot	Descartes	CAMP	Evocrash
Functional Suitability	Functional Completeness	4	4		5
	Functional Correctness	1	2		3
	Functional Appropriateness	1	2		1
Compatibility	Co-existence	5	5		5
Performance Efficiency	Time-behavior	1	2		1
Usability	Appropriateness recognisability	2	4		3
	Learnability	2	4		3
	Operability	1	4		3
Reliability	Maturity	2	3		2
Portability	Installability	2	4		3

11. Use Case : OW2

Introduction

In this section we present the context of the OW2 use case.

The STAMP use case aims at enhancing the quality of OW2 Java code base. OW2 is a non-profit organization which mission is to foster the growth of a portfolio of freely accessible open source software for enterprise information systems. The functional scope cover infrastructure software at large including middleware such as application servers, enterprise service bus, transaction monitor and portal, application platforms such as business intelligence, content management, identity management, cloud computing,



incident and event management (SIEM), asset management, business process management, etc. and the tools to develop and manage them. From a language point of view, although there is no exclusive, most of the code base is written in Java. Other languages such as Perl and Python are represented but it looks like the code base is going to be predominantly Java in the long term which is the reason why OW2 invests in quality and testing tools for Java.

The STAMP use case is a community endeavor which will be evaluated by the OW2 Technology council. As an association which job is to promote open source software, OW2 does not develop software by itself. OW2's code base grows by accretion of third party projects submitted by project leaders who wish to benefit from synergies with the community, promotional support from the organization and the critical mass afforded by the fact of belonging to OW2. OW2 has implemented a governance system that provides a framework to run community activities, manage the growth of the code base and verify the open source status of the projects. The growth of the code base is managed by the community of project leaders who are all members of the technology council. They accept new projects in the code base, manage their evolution through the OW2 project lifecycle and decide new technology orientations.

The STAMP use case will help enhance the project life cycle testing criteria category. The OW2 project lifecycle runs through three stages: incubation, mature and archive. To qualify as mature, a project must comply with a set of criteria defined by the technology council. Projects in incubation are either new projects or those that do not comply with mature projects criteria. Projects in archive are obsolete or deprecated projects. Mature projects represent OW2's main assets. The code base is currently comprised of more or less 100 projects, 30 of which are active mature projects, the rest being split between projects in incubation and projects in archive. Project maturity is determined by ten categories of criteria. Software testing is one of these categories, others include documentation, feature roadmapping, contributor management, etc.

Because OW2 has full control over its technical infrastructure it will leverage the STAMP use case to incorporate the new testing tools as a service to its community. Indeed, OW2 is an independent organization and independence is a key advantage in the world of open source software. It means OW2 is not subjected to any dominant industry power and not exposed to unexpected strategy shifts decided to maximise shareholders value. No IT giant will ever repropriate the OW2 code base – compare this with Nokia's decision to repropriate the Symbian operating system – nor acquire OW2 – compare with Microsoft's acquisition of GitHub. An essential element of this independence is that OW2 runs its own technical infrastructure. The technology council also provides supervision and recommendation as to the evolution of the OW2 technical infrastructure. Built around GitLab community edition, the infrastructure offers development teams some 15 services including source code management, issue tracking, continuous integration, contributor management, cloud deployment, wikis, mailing lists, etc. The infrastructure also includes software analysis tools such as SonarQube and ScanCode.

The STAMP use case will help integrate new quality tools into the OSCAR platform. Several years ago, OW2's board of directors and technology council approved the launch of a quality program to facilitate the promotion of the code base. These efforts target primarily the mature projects in the OW2 code base. The on-going enhancement of the technical infrastructure contributes to this objective. The quality program consists in sharing with project leaders the scores of data OW2's technical infrastructure generates on the projects it supports. OW2's quality-oriented resources are grouped under the OSCAR generic name. OSCAR stands for Open Source Capability Assessment Radar, it follows a previous program called Software Quality Assessment and Trustworthiness. OSCAR is both a platform of technology resources and a methodology for providing decision support results. OSCAR serves the dual purpose of helping improve the quality of the OW2 code base and facilitate its promotion on the market. OSCAR is the technical platform which provides the data helping the technology council manage the projects life cycles. OW2 has incorporated into its governance process a requirement that, for a project to be moved from incubation to mature, it must produce a report on the quality of its code and on its IP compliance. These results are used by the technology council



to decide on a project progress along the OW2 project lifecycle Having good quality code and a good reputation for code is essential for the success and growth of OW2. It is also essential for the growth of the downloads and the dissemination of the OW2 code.

The STAMP use case will contribute to making OW2 project more reliable and acceptable by conventional decision makers who need to be convinced of the quality of open source components.

While data provided by OSCAR is for developers, OW2 is currently developing a comprehensive approach targeted at conventional decision makers who are not necessarily open source supporters. The overall strategic objective is to make them perceive OW2 software as reliable as proprietary software. We have thus developed the OW2 Market Readiness Levels (OW2 MRL). It is a series of business-related situations that reflect the evolution of a project from initial software development to market leadership. The sequence takes into consideration changes that must occur for a project to transform itself from a purely technology endeavour to a market powerhouse. Because we replicate the NASA TRL scale, the OW2 MRL scale also has nine levels.

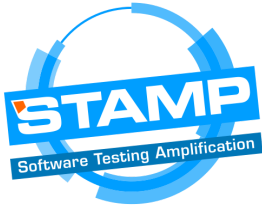
Validation objectives for the period

This part describes the validation-related activities conducted by OW2 in the period M6-M18 (April 2017-May 2018).

The STAMP use case is being developed as an extension of the OSCAR technical framework. The OSCAR platform is for now organized in four essential parts: an open source maturity model, code quality analysis resources, intellectual property analysis resources and evaluation models. Each part is instrumented by one or several tools.

- Open-source Maturity Model. The Open-source Maturity Model (OMM) is a maturity model and assessment methodology from the QualiPSo project. The OW2 OMM assessment template to be filled in by OW2 projects is available at OMM and is under constant evolution under the leadership of the OW2 Technology Council.
- Code quality analysis. Code quality analysis is provided by SonarQube, a static analysis solution covering a wide variety of languages including Java, Python, Erlang, C++. It implements the SQALE methodology to evaluate the technical debt of a project. STAMP will complement OSCAR's code quality analysis part.
- IP analysis. Intellectual property was provided by FOSSology which is currently being replaced by ScanCode. ScanCode is an open-source license compliance software system and toolkit. It allows us to run license, copyright, and export control scans from the command line or from a web user interface. ScanCode implements the SPDX standard – Software Package Data Exchange. Results of FOSSology and ScanCode applied to the OW2 projects are made available from the projects dashboards.
- Evaluation models. OW2 has defined a set of evaluation models used to compute the projects market readiness. Each model consists of normalization intervals, a license risk function, a quality risk function and an activeness risk function. The models we are currently using were created as a use case within the EU-funded RISCOSS project.

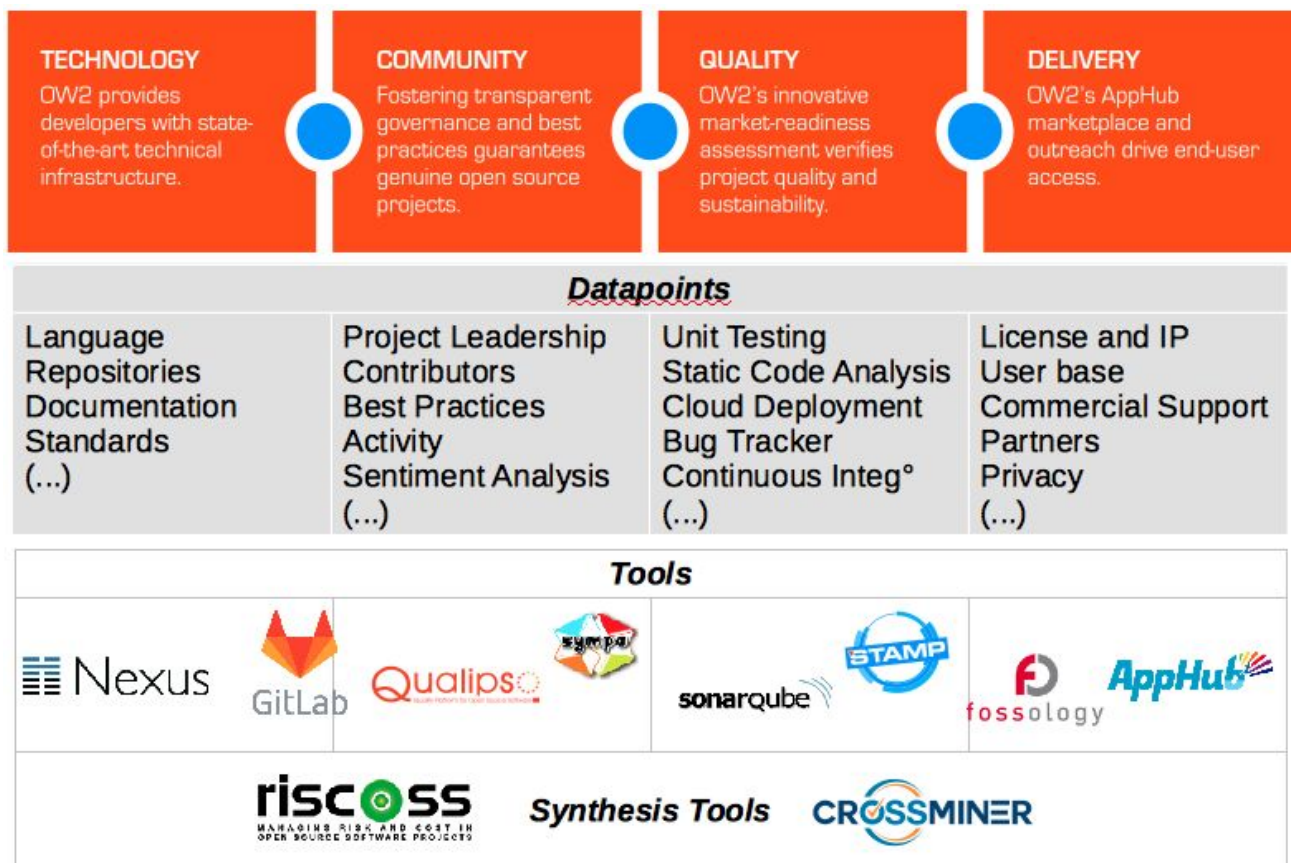
OSCAR is a platform that collects data from dozens of data points and combine them into models to provide value added information on projects. Datapoints are essentially associated with developers tools. Data are collected by means of dedicated scripts, called scrappers, that are applied to services such as GitLab, SonarQube and ScanCode. Scrappers are automatic and can be activated as often as necessary. At this stage they are programmed to be activated either routinely on a monthly basis or upon specific event such as a new commit or a new build. Automatic datapoints provide some 60% of data required by the OSCAR assessment methodology. Scrappers feed data into the OSCAR data collector which normalize them and inject them into the models. This sequence however is a development in progress and will be completed before the end of the project. Data collected manually are those that cannot be scrapped from the technical



infrastructure and have therefore to be either collected manually by OW2 or declared by project leaders. The method to collect this data is a form called Open Source Maturity Model (OMM). The OMM form is derived from QualiPSO, another EU-funded, we have adopted it to the needs and specifics of the OW2 approach.

In this context, the STAMP use case will provide additional data points to enhance the code quality analysis, provided by Sonarqube, and the continuous integration tools provided by Gitlab. We are also aiming at a fifth part: proposing tools to projects to enhance runtime quality. STAMP use case will provide tools to help projects in reproducing and fixing runtime crashes. The interest of STAMP and of its integration into the OW2 platform will be evaluated by the project leaders in collaboration with the OW2 Management Office, i.e. the team running the Consortium on a daily basis. The ultimate objective is to combine STAMP with the existing OSCAR platform so as to provide OW2 projects with enhanced testing tools at the code, configuration and runtime levels.

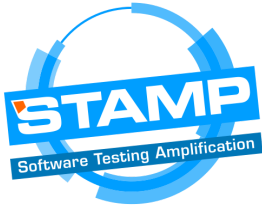
The exhibit below illustrates STAMP's positioning in the context of OW2's OSCAR platform.



An overview of the OSCAR platform

In order to achieve those objectives, we defined the following approach for this period:

- Learn how to use STAMP tools
- Apply those tools on one candidate project of OW2 code base
- Understand how we can collaborate with OW2 project leaders
- Evaluate how STAMP tools can be integrated into OW2 platform and methodology
- Evaluate how we can offer a testing service to OW2 project leaders



Validation work

Validation preparation

The OW2 use case consists in experimenting the STAMP components in the context of the OW2 quality program.

Since the OW2 team does not develop software by itself, the first preparation task consists in selecting and signing up the projects which will be part of the use case. We focused on four OW2 projects selected for their maturity level in particular in the area of quality assurance, the size of their customer base, their complexity, and the compatibility level of their underlying technologies with the ones supported by STAMP. The initially targeted projects are: JORAM, Lutece, Sat4j, ASM. This selection covers three application domains representative of the OW2 code base, namely machine-to-machine (JORAM), content management (Lutece) and software engineering (Sat4j and ASM).

OW2 assessment during this reporting period will target the evaluation of the following STAMP tools:

- DSpot: <https://github.com/STAMP-project/dspot>
- Descartes: <https://github.com/STAMP-project/pitest-descartes>
- Evocrash: <https://github.com/STAMP-project/EvoCrash>

Configuration tools were not evaluated during this period. We realized those tools needed to be in touch with the project leaders from the beginning and since we had in mind to learn the STAMP tools first by our own, we decided to focus on the other tools mentioned above and work on this one once the collaboration process with project leader established.

We focused during this period on the Sat4j project as we emphasized test amplification and Sat4j has a large JUnit tests base (Java unit tests).

Sat4j is a Java library for solving boolean satisfaction and optimization problems. It can solve SAT, MAXSAT, Pseudo-Boolean, Minimally Unsatisfiable Subset (MUS) problems. Being in Java, the promise is not to be the fastest one to solve those problems (a SAT solver in Java is about 3.25 times slower than its counterpart in C++), but to be full featured, robust, user friendly, and to follow Java design guidelines and code conventions (checked using static analysis of the source code).

Sat4j uses both Ant and Maven for integration. This project uses OW2 Gitlab service for continuous integration. During this period, we didn't focus on integrating STAMP tools into the CI, this will be done during the next period with the help of project leaders.

In order to validate the tools mentioned above, we decided to apply them on Sat4j project with the help of the developers and also with the help of Sat4j project leader when necessary.

Our work consisted in first place to clone the tools repository and understand how they work.

DSpot

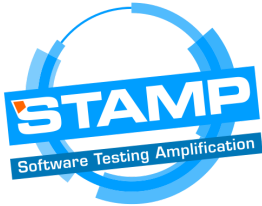
DSpot can be executed from the source code or from a Java Jar archive downloadable on Github page, but also from a Maven plugin (<https://github.com/STAMP-project/dspot/tree/master/dspot-maven>).

It requires to configure a property file for the target project.

Target project source is needed also, since DSpot relies on existing test suites.

Communication channel we used mainly for this tool was Github issues tab. It would not be pertinent to give the list of all issues, but we picked some with interesting exchanges and summarized them.

- <https://github.com/STAMP-project/dspot/issues/251>: on this issue, we couldn't launch the tool. It appeared we had difficulties to configure the property file. DSpot team gave precise indications on how to do it adequately.



- <https://github.com/STAMP-project/dspot/issues/264>: on this issue, we can see that when code is generated in a project building phase, DSpot was failing. This issue was fixed with a new release and by specifying a new property to the configuration file.
- <https://github.com/STAMP-project/dspot/issues/267>: on this issue, there was some problem with some classes in the project. In order to be able to use the tool in that specific case, DSpot team suggested to use a property that excludes classes during the process.
- <https://github.com/STAMP-project/dspot/issues/415>: on this issue, we can see that DSpot has problems to go to the end of new test generation. This issue is still open but we can see that DSpot team is very reactive when it comes to handling new issues.

Descartes

Descartes has to be installed from the source code via the Maven install command.

In order to run it, the target project Pom.xml file should be completed with Descartes parameters.

Target project source is needed also, since Descartes relies on existing test suites.

Communication channel we used mainly for this tool was email messages with Descartes lead developer and Sat4j project leader.

EvoCrash

EvoCrash can be both executed from the source code or from a Java Jar archive downloadable on Github page. In order to run it, it requires a crash log file with a stacktrace. Communication channel we used mainly for this tool was Skype, as we can see in this issue:

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

Reason is that tool is quite complex to run, so we asked EvoCrash lead developer to assist us in order to understand better how it works.

Validation result

DSpot

From each existing JUnit test files incorporated in the project source code, DSpot generated its own amplified test files. DSpot outputs consists in new automatically generated JUnit test files and reports about mutants killed. Those reports are declined in two versions: one text file and one JSON file.

The output text file (below) is a summary of the DSpot execution, we can find this information for instance:

- since DSpot is based on Pit, the first line is the result of Pit standard execution. We can see that Pit generated 357 mutations .
- DSpot, on top of that, added two new test cases. The results of those new tests is that this led to 15 new mutations. The details of those mutations is found in the corresponding JSON file.

```
===== REPORT =====
```

```
PitMutantScoreSelector:
```

```
The original test suite kills 357 mutants
```

```
The amplification results with 2 new tests
```

```
it kills 15 more mutants
```

The JSON report file provides the DSpot results for the six test cases in the test file. For each single mutation, we have the details concerning the line number in the code and the corresponding method. Here is below an extract from this report:

```
"nbMutantKilledOriginally": 0,  
"name": "org.sat4j.BugSAT17",  
"nbOriginalTestCases": 6,  
"testCases": [  
  {  
    "name": "testSingleLit",  
    "nbAssertionAdded": 2,  
    "nbInputAdded": 0,  
    "nbMutantKilled": 1,  
    "mutantsKilled": [  
      {  
        "ID": "org.pitest.mutationtest.engine.gregor.mutators.ArgumentPropagationMutator",  
        "lineNumber": 158,  
        "locationMethod": "ensure"  
      }  
    ]  
  },  
]
```

What we learnt is that DSpot is able to automatically generate new test cases. But we had trouble interpreting the results as we cannot see how it will help project leaders to improve their tests. So we were in touch with Sat4j project leader but didn't have any feedback on this point. As a lesson for this period, we understood we should emphasize the collaboration with project leaders since they are the ones who have the best knowledge of their code.

Descartes

Descartes experimentations output can be found on Github repositories:

<https://github.com/STAMP-project/descartes-experiments/tree/master/sat4j-core>

We had some issue to run Descartes on our own, Descartes lead developer helped us kindly to configure properly Pom.xml file.

Results were provided to Sat4j project leader and he was surprised of the results, he asked for some explanation. Descartes lead developer provided him hints in order to fix some tests and thus have a better regression detection.

One method was covered directly by only one test case to target a specific bug and avoid regression issues. The other method was covered indirectly by 68 different test cases. The lead developer considered the first method as helpful to realize that more assertions were needed in the covering test case. Consequently, he made one commit

(<https://gitlab.ow2.org/sat4j/sat4j/commit/46291e4d15a654477bd17b0ce905926d24e042ca>) to verify the behavior of this method.



The second pseudotested method was considered a bigger problem, because it implements certain key optimizations for better performance. No test case was difficult enough to witness the effect of such optimization in the test suite. Consequently, the developer made a new commit (<https://gitlab.ow2.org/sat4j/sat4j/commit/afab137a4c1a54219f3990713b4647ff84b8bfea>) with an additional, more difficult, test case to target the issue.

SAT4J and Auzthforce leaders were very supportive and provided very valuable feedback about each method. Results from meetings about pseudo-tested methods can be found on the project github:

<https://github.com/STAMP-project/descartes-experiments/blob/master/actionable-hints/sat4j-core/sample.md>

<https://github.com/STAMP-project/descartes-experiments/blob/master/actionable-hints/authzforce-core/sample.md>

On this tool, communication worked very well between Descartes lead developer and Sat4j project leader. And this led to new test cases in Sat4j project, which is promising for the future. We learnt during this period that although this communication is efficient, we have to pay close attention to monitor those exchanges, otherwise we would miss some information. And this information is very important to us since the goal of our use is to enhance our quality platform. Thus we need to identify the strong and weak points of each tool.

Evocrash

Evocrash needs a log file as input in order to generate new tests that replicate an exception. So we asked Sat4j project leader to provide us a crash log file sample.

The tool allows us to configure the depth of the exception: for instance, let say there are 15 nested exceptions. So we will be able to configure how deep we want the algorithm to go. Depending on this depth and the source code complexity, it can require some time to generate a new test. For instance, one day, we ran the tool all day long every 30 mn and we had a result at the end of the day. But on another day, we had a result after 15 mn. This is due to the randomness of the algorithm used to reproduce the exception.

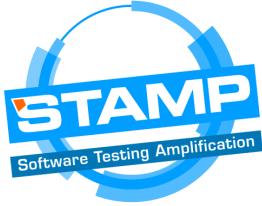
The new test generated by Evocrash includes the method that triggers the exception we are looking to reproduce, i.e. the same as in the crash log file provided. This new test is intended to help developers in fixing the bug by reproducing it. Unfortunately, at the moment we write this document, we didn't have updates from Sat4j project leader about these results.

What we learnt during this period on this tool is that for the moment the tool is not enough mature to let the project leaders use it alone. Although EvoCrash team helped us very kindly and very patiently to use the tool appropriately, we think that the limited capabilities of Evocrash to predict the time needed to run could be an obstacle for industrial use.

Tools presentation to projects

In February 2018, we did a presentation of STAMP to ten OW2 members project leaders, including Sat4j, Lutece, ASM and JORAM, our four targeted projects for this use case. The overall impression was very positive, everybody was enthusiastic to collaborate with us on these tools.

Our next step will be to focus on JORAM project, we have already started gathering informations about this project in the STAMP use case validation.



Quality Model

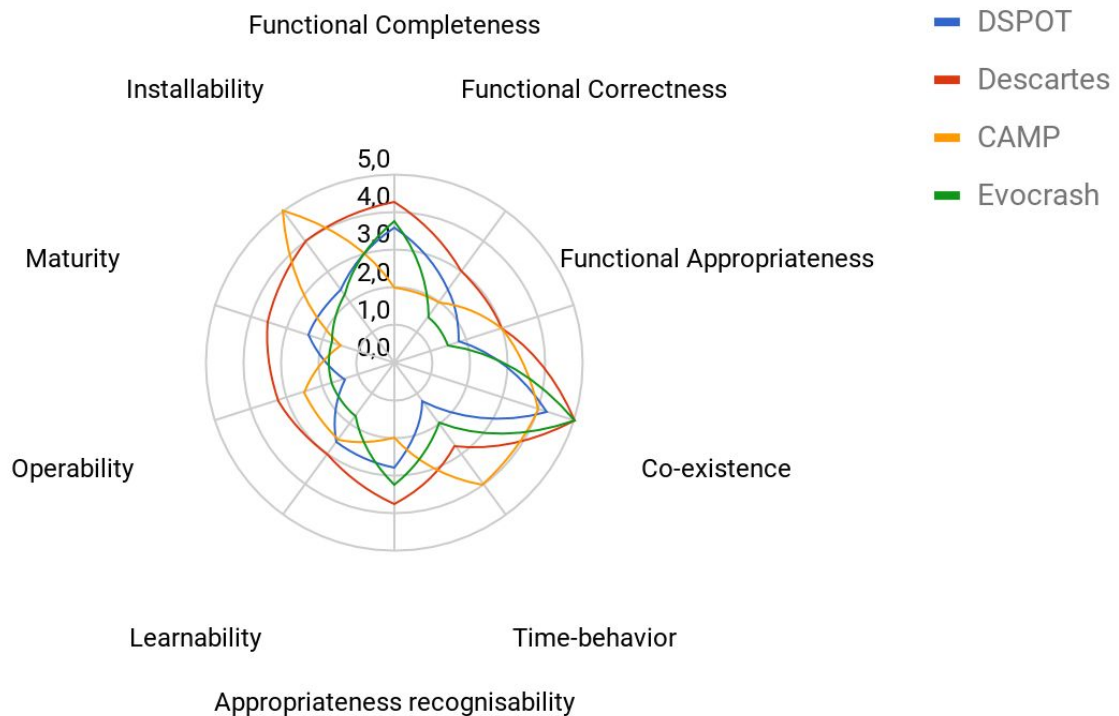
For each sub-characteristic and for each tool you used, provide your subjective degree of satisfaction (find definition for these characteristic in the appendix or https://edisciplinas.usp.br/pluginfile.php/294901/mod_resource/content/1/ISO%2025010%20-%20Quality%20Model.pdf) please use scale (0 : worst - 5 : best, ? for N/A)

Don't need to fill unused tool/column

Characteristic	Sub-characteristic	DSpot	Descartes	CAMP	Evocrash
Functional Suitability	Functional Completeness	3	4		3
	Functional Correctness	2	3		1
	Functional Appropriateness	2	2		3
Compatibility	Co-existence	5	5		5
Performance Efficiency	Time-behavior	1	2		2
Usability	Appropriateness recognisability	1	3		3
	Learnability	1	1		1
	Operability	1	2		1
Reliability	Maturity	2	3		2
Portability	Installability	1	3		1

12. Assessment summary and metrics

In this section, we summarize the assessments and recommendation expressed by the uses cases partner. First we present the global numerical evaluation based on our quality model assessment, then we detail all recommendation proposed by STAMP tools users.



In the above diagram, we present the result of the average quality evaluation conducted by use case partners. The DSpot tool was evaluated by the five partners, Descartes and Evocrash were evaluated by four partners, while CAMP was only evaluated twice. By coincidence, the profile obtained by DSpot and Evocrash are quite similar. They demonstrate a very good (>4) co-existence through their integration into existing testing tools, a good (>3.5) functional completeness, and an acceptable (>2.5) appropriateness recognisability, meaning that these tools target useful features. On the other hand, other characteristics are evaluated below average. Especially, functional correctness (1.5) and functional appropriateness (1.5) of Evocrash is impacted by un-usefulness of results obtained, while DSpot operability (1.4) and time behavior (1.3) is impacted by difficulties in usage. There is only two (partial) evaluation of CAMP, so it's hard to derive accurate information, but several effort must be conducted to obtain a meaningful tool. Descartes tool quality has an overall better profile than the others, with all criterias evaluated good enough (≥ 3). In particular, the tool have a very good instability and functional completeness. Points of improvement are learnability, functional completeness and functional appropriateness.

Detailed recommendations

DSpot

Concerning DSpot, the recommendations of the partner are the following.

- Documentation: For the versions tested, more documentation was needed. Knowing how to correctly configure the tool was a hurdle. However, both the tool and the documentation has been advancing since then. What would be useful is some information on what to expect when running the tool – how long it takes and what results are likely for different types and sizes of projects.
- Time efficiency: The tools seems too slow to execute
- Accuracy: DSpot finds very few tests to add. This is the biggest issue at this point.
- Usability: Once properly configured, user have hard time understanding how to exploit those new tests. Results are not presented in a clearly understandable way, as such they are not readily useful.

Some work is required in order to get the functional value added. We think it would be nice if all reports were compiled in one unified report without having to browse through each individual file.

Descartes

Concerning Descartes, the recommendations of the partner are the following.

- Accuracy: Descartes is already working well, and we have little to add. We know that it produces fewer mutants and may therefore give a more coarse-grained score than the default PIT engine, so it would be interesting to know more about this (can we say something general about its accuracy?).
- Time efficiency: Descartes was perceived as too slow to execute. The timeouts are a big part of the slowness. Maybe record timeouts so that next time Descartes is executed they're not executed again, i.e. have a learning feature. However that wouldn't integrate that well with a local Maven build on developers matching. But it would when executed on the CI, which is very interesting.
- Usability: Descartes report should be modified to highlight/make it clear what are the strong pseudo-tested methods since they are the interesting code to analyse. Missing added-value as is, since it requires a human to analyze results. However combined with a fail build on low mutation score strategy it becomes more interesting. Same would apply for a GitHub PR-based check.
- Operability: as OW2 said, we had some trouble configuring Descartes. We think there is room for improvement in that part. Once properly configured, results are pertinent and that use case showed perfectly how it could help a project to have a better code coverage.

CAMP

Concerning CAMP, the recommendations of the partner are the following.

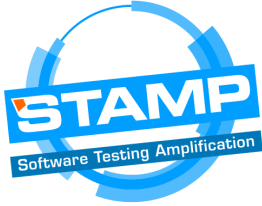
- Usability: Improve documentation and examples would be the best way to improve CAMP usability.
- Functional completeness: CAMP is also perceived as quite promising for automating the testing of a system under different deployment configurations, but it is unclear if CAMP will be able to search for and generate optimal configurations, provided that non-functional models (i.e. for performance, etc) are given as input.

EvoCrash

Concerning EvoCrash, the recommendations of the partner are the following.

- Usability: Generated tests cannot be committed as is; they're usually not done at the right level and need rewriting to be understandable at a higher more global level. Would be nice if the generated tests could mimic the styles of existing tests.
- Operability: The pull request <https://github.com/STAMP-project/EvoCrash-demo/pull/11> solve several issues that Activeeon and XWiki encountered. Moreover testing and providing feedback on this pull request would also be great for quality and reliability. Also, error messages generated by the tool could be more informative.
- Efficiency: By default, according to a range of target frames, statically predefined, or defined by the user, EvoCrash could generate a bag of tests. This tool generates new tests with an algorithm using randomness. Results can take some time, and sometimes there could be no results, depending on the complexity of the project to be analyzed. Besides being frustrating, it can be really time consuming. Improvements that provide an idea of the running time, or a progression status; as well as an indication of the potential buggy frame to select as target frame for the generation are welcome.
- Documentation: the GitHub documentation should have a "Getting started" which works on Windows, or it should point out the issues running on Windows. It would also be useful to have some information on what to expect when running the tool – how long it takes and what results are likely for different cases. Some partners had some trouble running the getting started, the overall impression is that tool is not easy to use. The work is still in progress, configuration for instance is not easy at beginning, it requires some technical skill.

Other recommendation



Horizon 2020
Communications Networks, Content & Technology
Cloud & Software Research & Innovation Action
Grant agreement n° 731529

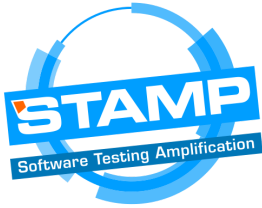
Coverage reports are important to evaluate software quality, and impact of STAMP tools on this quality. Concerning comparing coverage reports, it has been noticed it's not enough to have a local strategy (i.e. measure coverage at the module level). There are wide measure variations when measuring the coverage globally. This is caused by several aspects: functional tests exercising code indirectly, modules being removed, new modules being added. Thus a global coverage strategy is required with associated tooling (e.g. a Jenkins pipeline script taking those points into account).

13. Appendix : KPIs

In this appendix we report for each use case the KPI as measured in D5.1. Due to the appropriation and progression of STAMP tools in this first period, the KPI does not demonstrate a significant impact of the project on the use cases. However, this KPI need to be updated to observe progression in the next period.

Activeeon

Use Case Provider	Activeeon	
Collection Number	1 (from D5.1)	2 (26 April 18)
Tool used for measurement	SonarQube	SonarCloud
Collected value	30.6%	24.7%
K02 - Less flakey tests		
Total test count	1225	1251
Number of test runs in sample	1225	1251
Number of tests which failed and then passed	0	0
K03 - Faster tests		
Time taken during execution of tests	14 min 32 seconds	14 min 45 seconds
Test coverage (number of lines of code)	23373 (76633 * 0.305)	21618 (87526 * 0.247)
K04 - More unique invocation traces		
Number of invocation traces between services	N/A	N/A
K05 - System-specific bugs		
Time period of collection	N/A	N/A
Number of configuration-specific bugs detected	N/A	N/A
K06 - Faster deployment for testing		
Time required to deploy software for testing	7 min 40 seconds	8 min
K07 - Shorter logs		



Description of deployment for sampling	N/A	N/A
Log file size	N/A	N/A
K08 - More crash replicating test cases		
Number of observed automated test cases which replicate crashes	0	0
K09 - More crash replicating test cases		
Total number of automated test cases	1225	1251
Number of observed automated test cases which replicate crashes	0	0

ATOS

The initial set of KPIs collected in D5.1 [3] were collected for the SUPERSEDE DAPPLE (Dynamic Adaptation Platform) component. However, the way this component is built is incompatible with the STAMP tools: DSpot and Descartes. Building DAPPLE requires the usage of the Maven Tycho plugin since most of its subcomponents are Eclipse plugin based, but the Tycho plugin does not generate pom.xml descriptors for modules in the file-system, as required by the STAMP tools. Therefore for measuring KPI progress, we are now adopting the SUPERSEDE IF component, which can be built both with both Maven and Gradle.

The STAMP tools evaluated by Atos UC, namely Descartes and DSpot have direct or indirect impact in some KPIs collected in D5.1. Following table collects measured metrics for these KPIs, before and after applying STAMP tools on Atos SUPERSEDE IF code base.

Use Case Provider	ATOS	
Collection Number	SUPERSEDE IF (code base May 18)	SUPERSEDE IF (amplified test suite and code base May 18)
K01 - More execution paths		
Tool used for measurement	Clover Maven plugin 4.2.0	Clover Maven plugin 4.2.0
Collected value	49.2% (over total code base)	50% (over total code base)
K03 - Faster tests		
Tool used for measurement	Clover Maven plugin	Clover Maven plugin

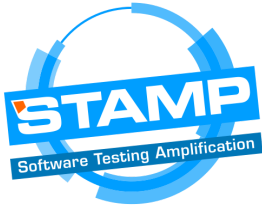


	4.2.0	4.2.0
Time taken during execution of tests	03:53 min time per test: 1.62 sec	15:44 min time per test: 1.51 sec
Test coverage (number of lines of code)	49.2% 280 classes, 3,636 / 7,385 elements LOC: 20,073 NCLOC: 14,005	50% 280 classes, 3,693 / 7,385 elements LOC: 20,073 NCLOC: 14,005

TellU

Due to the heavy refactoring of the TelluCloud code, metrics collected at the start of the project are not comparable to later metrics. Of the three selected new projects, metrics have been collected twice for TelluLib, listed here. Metrics have been collected once for Core and Filterstore, and these projects will be included in future KPI progression. Also note that the K04 numbers are not directly comparable.

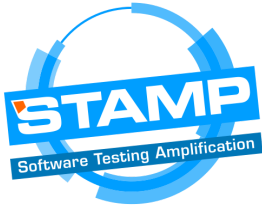
Use Case Provider	TellU	
Collection Number	1	2
K01 - More execution paths		
Tool used for measurement	Clover	
Collected value	34.0%	51.9%
K02 - Less flakey tests		
Total test count	49	75
Number of test runs in sample	5	5
Number of tests which failed and then passed	0	0
K03 - Faster tests		
Time taken during execution of tests	4.3 seconds	4.6 seconds
Test coverage (number of lines of code)	2690	4254
K04 - More unique invocation traces		
Number of invocation traces between services	27	7



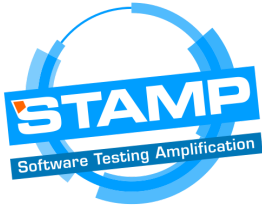
K05 - System-specific bugs		
Time period of collection	N/A	N/A
Number of configuration-specific bugs detected	N/A	N/A
K06 - Faster deployment for testing		
Time required to deploy software for testing	40 minutes	40 minutes
K07 - Shorter logs		
Description of deployment for sampling	N/A	N/A
Log file size	N/A	N/A
K08 - More crash replicating test cases		
Number of observed automated test cases which replicate crashes	0	0
K09 - More crash replicating test cases (% of suite)		
Total number of automated test cases	49	75
Number of observed automated test cases which replicate crashes	0	0

XWiki

Use Case Provider	XWiki	
Collection Number	2016-12-20	2017-11-09
K01 - More execution paths		
Tool used for measurement	Clover	
Collected value	65.29% (2016-12-20)	68.59% (2017-11-09)
K02 - Less flakey tests		



Total test count	9962	10416
Number of test runs in sample	N/A	N/A
Number of tests which failed and then passed	6	4
K03 - Faster tests		
Time taken during execution of tests	8257 seconds	7679 seconds
LOC / time	$736419/8257 = 89.18$ lines/s for 9962 tests	$639588/7679 = 83.28$ lines/s for 10416 tests
K04 - More unique invocation traces		
Number of invocation traces between services	N/A	N/A
K05 - System-specific bugs		
Time period of collection	1 year (2016)	1 year (2017)
Number of configuration-specific bugs detected	139 out of 4967 config tests = 2.79%	294 out of 6626 config tests = 4.43%
K06 - Faster deployment for testing		
Time required to deploy software for testing	2 hours	5 minutes
K07 - Shorter logs		
Description of deployment for sampling	N/A	N/A
Log file size	N/A	N/A
K08 - More crash replicating test cases (See: Note 7.8.1)		
Number of observed automated test cases which replicate crashes	3	5



K09 - More crash replicating test cases (% of suite)		
Total number of automated test cases	9962	10416
Number of observed automated test cases which replicate crashes	0.03%	0.04%

OW2

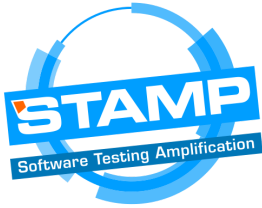
Project name	Sat4j (To be done)	
Collection Number		2
K01 - More execution paths		
Tool used for measurement		
Collected value		
K02 - Less flakey tests		
Total test count		
Number of test runs in sample		
Number of tests which failed and then passed		
K03 - Faster tests		
Time taken during execution of tests		
Test coverage (number of lines of code)		
K04 - More unique invocation traces		
Number of invocation traces between services		



K05 - System-specific bugs		
Time period of collection		
Number of configuration-specific bugs detected		
K06 - Faster deployment for testing		
Time required to deploy software for testing		
K07 - Shorter logs		
Description of deployment for sampling		
Log file size		
K08 - More crash replicating test cases		
Number of observed automated test cases which replicate crashes		
K09 - More crash replicating test cases (% of suite)		
Total number of automated test cases		
Number of observed automated test cases which replicate crashes		

Appendix : Quality Model description (ISO/IEC 25010)

Characteristic	Sub-characteristic	Definition
Functional Suitability	Functional Completeness	degree to which the set of functions covers all the specified tasks and user objectives.
	Functional Correctness	degree to which the functions provides the correct results with the needed degree of precision.
	Functional	degree to which the functions facilitate the accomplishment of



	Appropriateness	specified tasks and objectives.
Compatibility	Co-existence	degree to which a product can perform its required functions efficiently while sharing a common environment and resources with other products, without detrimental impact on any other product.
Performance Efficiency	Time-behavior	degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements.
Usability	Appropriateness recognisability	degree to which users can recognize whether a product or system is appropriate for their needs.
	Learnability	degree to which a product or system enables the user to learn how to use it with effectiveness, efficiency in emergency situations.
	Operability	degree to which a product or system is easy to operate, control and appropriate to use.
Reliability	Maturity	degree to which a system, product or component meets needs for reliability under normal operation.
Portability	Installability	degree of effectiveness and efficiency in which a product or system can be successfully installed and/or uninstalled in a specified environment.