



STAMP

Deliverable D4.3

**Second public version of API
and implementation of ser-
vices and courseware**

Activeeon
SCALE BEYOND LIMITS

Atos

ENGINEERING

Inria
INVENTORS FOR THE DIGITAL WORLD

KTH
VETENSKAP
OCH KONST

OW2
SINTEF

tell.u

TU Delft
X-WIKI
THE BEST WAY TO ORGANIZE INFORMATION

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

Deliverable Number	:	D4.3
Title of Deliverable	:	Second public version of API and implementation of services and courseware
Dissemination Level	:	Public
Version	:	1.2
Latest version	:	https://github.com/STAMP-project/docs-forum/blob/master/docs/d43_second_api_public_version_services_courseware.pdf
Contractual Delivery Date	:	M24 November, 30 2018
Contributing WPs	:	WP 4
Editor(s)	:	Daniele Gagliardi, ENG
Author(s)	:	Daniele Gagliardi, ENG Valentina Di Giacomo, ENG Luca Andreatta, ENG Nicola Bertazzo, ENG Jesús Gorroñogoitia Cruz, ATOS Ricardo José Tejada, ATOS Mael Audren, AEON Caroline Landry, INRIA
Reviewer(s)	:	Benoit Baudry, KTH Brice Morin, SINTEF

Abstract

This deliverable reports on the second release of STAMP components developed to integrate STAMP amplification services within developers' and DevOps tool-chains. The first part of the report describes the CI/CD reference scenario, with a complete working example of DSpot usage within the GitFlow branching model. The CI/CD scenario references also the usage of the other three STAMP tools (Descartes, CAMP and Botsing) within it. The second part of this report contains a description of the current state-of-the-art of STAMP ecosystem, a set of independent components that let STAMP adopters to use its novel features within their toolboxes. Each developed component has its own documentation to understand its usage and apply it within software lifecycle processes

Keyword List

Continuous Integration, Continuous Delivery, CI/CD Server, Issue Tracker, GitFlow, Branching Model, Pull Request

Revision History

Version	Type of Change	Author(s)
0.1	First draft	Daniele Gagliardi, ENG Caroline Landry, INRIA
0.2	Added missing picture about PR content	Daniele Gagliardi, ENG
0.3	Aligned content with feedback provided by internal reviewers	Daniele Gagliardi, ENG Valentina Di Giacomo, ENG Brice Morin, SINTEF Benoit Baudry, KTH
0.5	Second draft	Daniele Gagliardi, ENG
1.0	Converted in \LaTeX	Daniele Gagliardi, ENG
1.1	Alternate description of ecosystem. Added "future work" sections. Shortened the abstract and expanded the introduction	Daniele Gagliardi, ENG Nicola Bertazzo, ENG Luca Andreatta, ENG
1.2	Included comments and feedback from reviewers	Daniele Gagliardi, ENG Valentina Di Giacomo, ENG Brice Morin, SINTEF
1.3	Final release	Daniele Gagliardi, ENG Benoit Baudry, KTH Jesús Gorroñoitia Cruz, ATOS Mael Audren, AEON

Contents

List of Figures	8
List of Tables	9
1 Introduction	10
1.1 Relation to WP4 tasks	10
1.2 Participants contribution to WP4	11
2 The CI/CD Scenario	13
2.1 Introduction	13
2.2 STAMP core tools	13
2.3 The CI/CD landscape	14
2.4 GitFlow	16
2.5 STAMP Reference CI Scenario	17
2.5.1 Descartes: test your tests within CI/CD	18
2.5.2 DSpot: amplify your tests within CI/CD	20
2.5.3 CAMP: amplify your test configurations within CI/CD	26
2.5.4 Botsing: reproduce automatically production crashes within CI/CD	27
3 STAMP ecosystem: state of the art	31
3.1 DSpot	32
3.1.1 Build Systems	32
3.1.2 Developer Productivity Tools	32
3.1.3 CI/CD Systems	36
3.1.4 STAMP Tooling API	37
3.1.5 Documentation	37
3.2 Descartes	38
3.2.1 Build Systems	38
3.2.2 Developer Productivity Tools	39
3.2.3 Source code repositories	40
3.2.4 CI/CD Systems	40
3.2.5 Documentation	42
3.3 CAMP	43
3.3.1 Containerization	43
3.3.2 Documentation	43
3.4 Botsing	43
3.4.1 Build Systems	43
3.4.2 Developer Productivity Tools	44
3.5 Future work	45



4	Conclusion	46
4.1	CI/CD scenario	46
4.2	Ecosystem	46
4.3	tutorials, courseware and documentation	46

Acronyms

EC	European Commission
CI	Continuous Integration
CD	Continuous Delivery
API	Application Programming Interface
QA	Quality Assurance
WP	Work Package

List of Figures

2.1	STAMP and DevOps	13
2.2	Typical CI/CD configuration. The CI/CD server is at the core of automation . .	14
2.3	Gitflow branching model (source: https://nvie.com/img/git-model@2x.png)	16
2.4	STAMP CI/CD Integration scenario.	18
2.5	Detailed report about Descartes execution in a Jenkins build	19
2.6	Descartes assessment over the time	20
2.7	Branching model with explicit branching to support test amplification.	21
2.8	Jenkins pipeline containing a test amplification stage.	22
2.9	Pipeline with test amplification execution in the Develop branch	23
2.10	Detailed steps for the Pull Request stage	24
2.11	Pipeline that skips test amplification execution outside the "test amplification" branch.	24
2.12	Pull Request details opened automatically by Jenkins and containing amplified test cases.	25
2.13	Pull Request content: test cases has been amplified with several new assertions.	26
2.14	Jenkins pipeline containing a test configuration amplification stage.	26
2.15	A crash occurring during operations triggers the bug fixing process within devel- opment.	27
2.16	Gitflow branching model to manage hot-fixes (https://nvie.com/posts/a-successful-git-branching-model/)	28
2.17	Bug triage automation Botsing flow within the CI/CD scenario.	29
3.1	STAMP ecosystem	32
3.2	STAMP IDE repository configuration	33
3.3	STAMP IDE Plugins installation	34
3.4	STAMP IDE menu	35
3.5	STAMP IDE: DSpot configuration wizard	35
3.6	STAMP IDE: DSpot results view	36
3.7	Jenkins: configuring a freestyle job to run DSpot	37
3.8	Descartes output views showing the PIT mutation score report and the Descartes issues report.	39
3.9	Descartes detailed analysis within Jenkins dashboard for the current build. . . .	41
3.10	Jenkins Descartes reports: build trends.	42
3.11	STAMP IDE: Botsing configuration wizard	44

List of Tables

1.1	WP4 tasks reported in this deliverable	11
1.2	WP4 participants and their activities	12
3.1	PitMP compatibility	38

Chapter 1

Introduction

STAMP main tools have been greatly consolidated within the first half of the project. This resulted in a clearer vision about how to integrate them within development tool-chains, and let us take appropriate decisions to set the integration priorities, having in mind the most natural context where STAMP can reach its full potential: DevOps. This report presents the second release of STAMP components developed to integrate STAMP amplification services within developers' and DevOps toolchains, along with documentation and course-ware available to understand their usage and apply them within software life-cycle processes. The reader will be presented with a description of a Continuous Integration Scenario, in which several components, orchestrated by a CI/CD server, cooperate to build software with an increased quality. Within this architecture, we will focus on the Gitflow branching model to provide a possible effective way to use STAMP within a widely adopted branching model among developer communities. This reference architecture can be expanded and varied, thanks to the fact that STAMP ecosystem is made of several services and plugins that facilitate the integration in different development contexts. This scenario leverages the idea to provide a reference architecture based on the concept of independent micro-services and at the same time aligning the STAMP architecture to the most common CI/CD scenarios used today.

In the second part of this deliverable we provide a description of the second version of STAMP integration status, starting with a state-of-the-art of the four STAMP tools and the ecosystem that we're building around them. Since December 2017, we developed other plugins and extension to ease the usage of STAMP tools in different tool-chains, leading to the definition of a consistent STAMP ecosystem: Maven and Gradle have been confirmed to be the most relevant build systems in the Java development world and for this reason we invested effort to let developers use STAMP features as ordinary Maven goals and Gradle tasks. Moreover we developed other plugins to use STAMP amplification features within CI/CD tools, focusing on Jenkins and GitHub CI platform. We also continued IDE integration development, extending the Eclipse IDE support to STAMP tools. Regarding documentation and course-ware, we kept on writing wiki pages on GitHub repository and on the official web site. Currently we are defining contents for a STAMP Cookbook to provide developers with guidelines to setup up properly and leverage STAMP features, starting from scenarios described in this report. The tools and results presented here are follow ups on the tools and results presented 10 months ago in deliverable D4.2. These tools are presented in chapter 3.

1.1 Relation to WP4 tasks

In Table 1.1 we summarize how the software assets developed since now and results reported in this deliverable relate to the 4 tasks of WP4.

Table 1.1: WP4 tasks reported in this deliverable

Task 4.1	This task, related to setting up and maintaining the STAMP project collaborative platform, has been extended at M36 with the last amendment. In this deliverable we enriched the collaborative platform, instantiating a Jenkins instance on a dedicated Virtual Machine to validate the CI/CD scenario
Task 4.2	We have defined the CI/CD integration architecture to support test amplification practices in a typical DevOps toolchain
Task 4.3	We built new plugins to keep on integrating STAMP in developers toolboxes
Task 4.4	We wrote the documentation needed to use STAMP tools and integration, in form of wiki pages on the official STAMP GitHub repository

1.2 Participants contribution to WP4

All the work done within the WP4 is the result of the collaboration of several STAMP project partners: it is a natural consequence of the fact that WP4 scope is about integrating STAMP tools. In Table 1.2 we summarize how we collaborated to achieve results presented in the current report.

Table 1.2: WP4 participants and their activities

INRIA, KTH	defined the CI/CD scenario as the reference scenario for integration activities (see section 2.5)
XWIKI	provided expertise in Jenkins usage, to define better the "amplify your tests" pipeline (see subsection 2.5.2)
ATOS	fully implemented the STAMP IDE (see subsection 3.1.2, subsection 3.2.2 and subsection 3.4.2)
Ac- tiveEon	provided expertise in Gradle build system and developed Gradle integration for Botsing (see subsection 3.4.1)
OW2	collaborated to setup the reference Jenkins instance in a dedicated virtual machine, within the STAMP collaborative platform (see subsection 2.5.2)
INRIA, SINTEF, TUD	provided support to understand how to embed respectively DSpot & Descartes, CAMP, Botsing in integration targets: build systems, IDEs, CI/CD servers, etc (see chapter 3)
ENG	defined (see chapter 2) the STAMP CI/CD reference architecture, configured the CI/CD server, implemented the "Amplify your tests" pipeline, developed (see chapter 3) the Jenkins plugins for DSpot and Descartes, Maven plugin for DSpot and Botsing, added support for Gradle projects within DSpot, wrote documentation and setup a new virtual machine with several preconfigured STAMP tools, and did several maintenance activities on the collaborative platform (installing several Jenkins plugins needed for the CI/CD scenario, investigating on a brute-force attack against the SSH service, and related cleanup activities due to running out of the disk space)

Chapter 2

The CI/CD Scenario

2.1 Introduction

This chapter provides a detailed description for the integration of STAMP with common CI/CD tools such as Maven, Jenkins and GitHub, in order to build a DevOps toolchain enhanced with test amplification capabilities. Each of the three STAMP amplification services finds its natural place within DevOps cycle: the objective of this chapter is to be a reference guide to implement the DevOps STAMP-enhanced cycle:

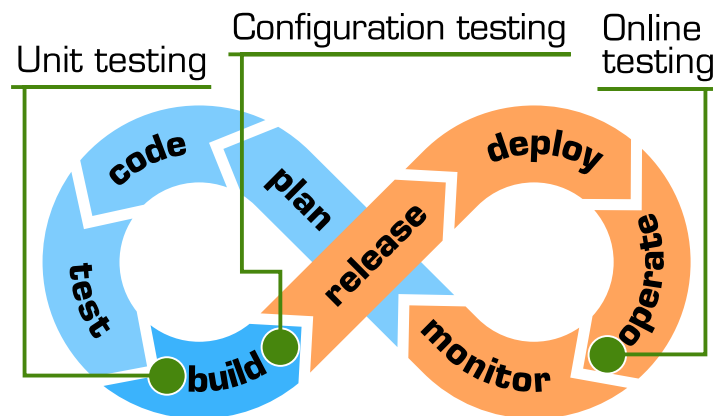


Figure 2.1: STAMP and DevOps

2.2 STAMP core tools

When referring to STAMP tools in this document, we refer to the 4 technical components for test amplification developed in Work-packages 1, 2, 3:

- DSpot: a tool that generates missing assertions in JUnit tests.
<https://github.com/STAMP-project/dspot>
- Descartes: evaluates the capability of a suite to detect bugs using extreme mutation testing.
<https://github.com/STAMP-project/pitest-descartes>

- CAMP: automatically generates and deploy a number of diverse testing configurations.
<https://github.com/STAMP-project/camp>
- Botsing: a Java framework to automatically generate crash reproducing test cases.e
<https://github.com/STAMP-project/botsing>

2.3 The CI/CD landscape

At the core of DevOps there is a huge emphasis on automation: build, deployment, testing need to be automated as much as possible, in order to have smooth integrated flow which cycles and produces new code continuously. A more appropriate statement should be “continuous development” in order to take explicitly into account the fundamental role of who actually produces software: the developers. No automatic source code generation can replace human capabilities to develop working solutions to concrete needs, but automation can “amplify” human capabilities to produce more reliable software. The following picture shows a typical CI infrastructure:

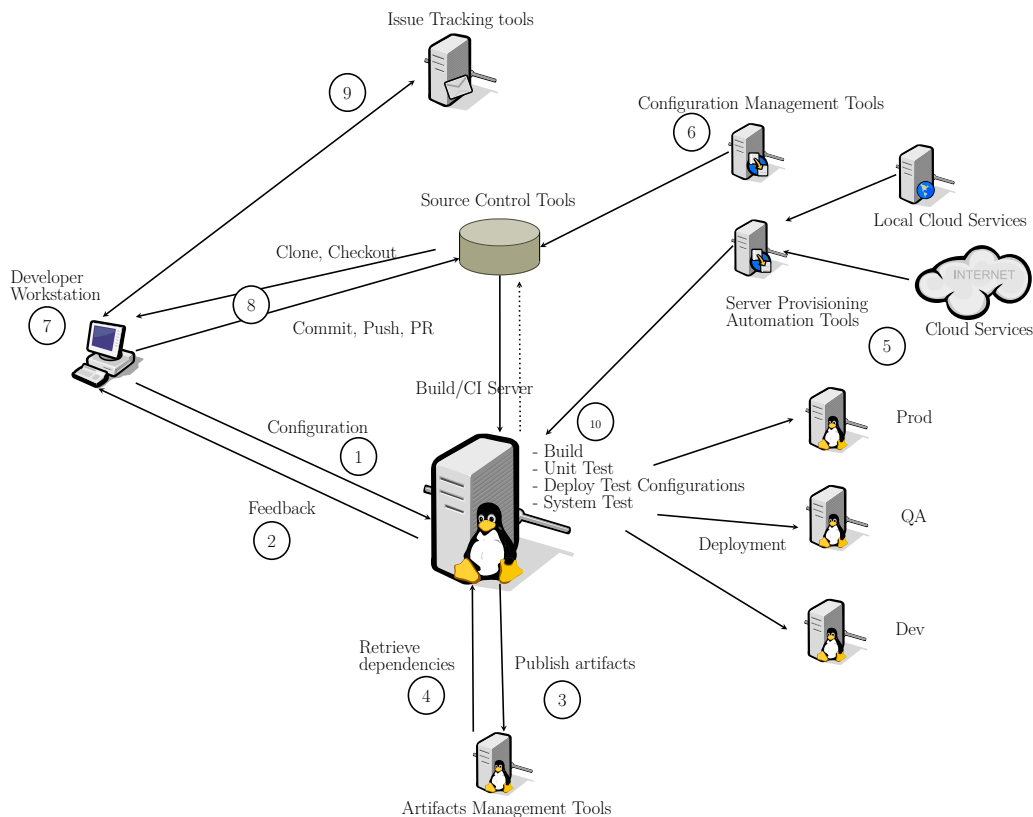


Figure 2.2: Typical CI/CD configuration. The CI/CD server is at the core of automation

The **Build/CI Server** has the critical role of providing and supervising the integrated automation of all the tasks that contribute to build the software:

1. Developers provide the CI Server with specific **configurations** that support their automated processes
2. Developers receive **feedback** from it, in form of reports, dashboards and software artifacts

3. Software artifacts are stored in **artifacts repositories**
4. Artifacts repositories provide CI/CD server with the dependencies needed to build software
5. Developers also configure and integrate tools to provide target systems where the software will run (**test configuration provisioning**)
6. Developers will manage target systems with **configuration management tools**, using "infrastructure as code" practices
7. Developers will have **their own productivity tools** to write source code and tests
8. Developers will store them in **source code repositories** which will keep track of any change
9. **Issue trackers** are used by developers to track all tasks and user stories that model system requirements, as well as bugs that arise from testing and from production
10. The Build/CI Server is configured to execute several jobs which run periodically or are triggered by specific events (commits, push, pull requests, etc). Typical jobs are:
 - (a) build the software
 - (b) execute unit and integration tests
 - (c) publish artifacts
 - (d) instantiate target server (QA, Test, Prod), deploy software on them and execute system tests

This scenario can be implemented with:

1. a Jenkins CI instance to support CI/CD pipelines (today Jenkins is THE CI/CD server: it is an open source solution with a huge marketplace with plugins that extends its functionalities, with a vibrant community who contributes daily to its improvement and a well-documented programming model);
2. a GitHub repository to host source code (GitHub is a cloud service that wraps the git version control system, along with a plethora of external services which provides the platform with build systems, code and test coverage analyzers, etc. It is one of the biggest developer community in the world. It is open to integration thanks to the mechanism of GitHub Apps);
3. Atlassian Jira as the reference issue tracker (the reason why we chose it was that Jira is probably the most used issue tracker in the world, it's well-known by developers communities, and moreover, it has a powerful SDK to build extensions for integrating it with almost every tool available. See <https://developer.atlassian.com/server/framework/atlassian-sdk/>).
4. Docker as the containerization technology to manage services a in "Infrastructure as Code" context.

Around this core, several different choices can be made for the artifacts repository, configuration management and provisioning tools, etc.

2.4 GitFlow

Another important aspect in collaborative development is the approach to source code versioning. When multiple developers collaborate to a software project, it is a best practice to create several branches on the source code repository, to let each developer work on a specific feature or bug-fix, isolating new development from finished work. This collaborative model requires a branching model to keep collaboration working effectively. One of the most adopted branching models is GitFlow, created by Vincent Driessen (<https://nvie.com/posts/a-successful-git-branching-model/>), It has attracted a lot of attention because it is very well suited to collaboration and scaling the development team. The following picture summarizes this branching model:

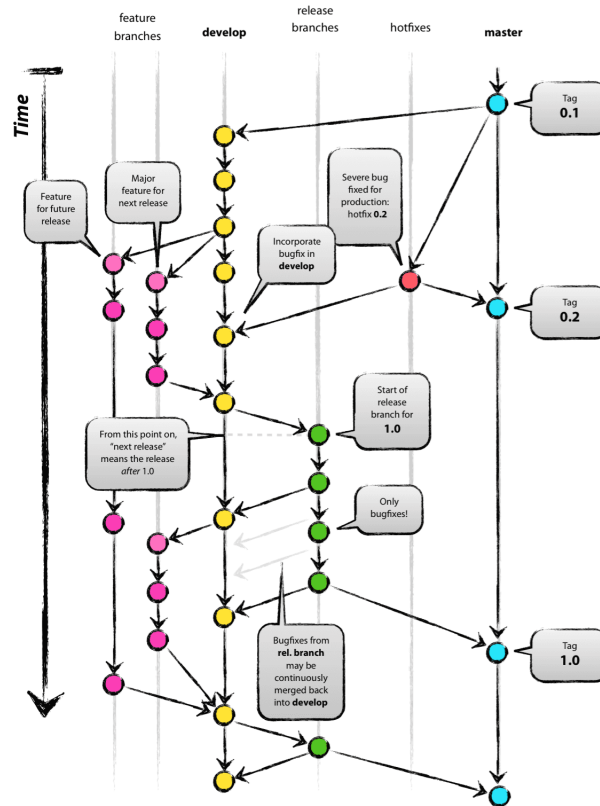


Figure 2.3: Gitflow branching model
(source: <https://nvie.com/img/git-model@2x.png>)

Gitflow makes parallel development very easy, by isolating new development from finished work. New development (such as features, bug fixes, etc) is done in dedicated branches, and is merged back into the main code branch when each developer is happy with the code he wrote. **Feature branches** also make it easier for two or more developers to collaborate on the same feature, because **each feature branch is a sandbox** where the only changes are the changes necessary to get the new feature working. As new development is completed, it gets merged back into the **develop branch**, which is a staging area for all completed features that haven't yet been released. So when the next release is branched off of develop, it will automatically contain all of the new code that has been finished. GitFlow supports hot-fix branches - branches made from a tagged release. You can use these to make an emergency change, knowing that

the hot-fix will only contain your emergency fix. There's no risk that you'll accidentally merge in new development at the same time.

2.5 STAMP Reference CI Scenario

Starting from the reference CI/CD scenario and the GitFlow branching model described in the previous section, we defined several STAMP integration points to provide the CI/CD scenario with STAMP unique features:

- (A) test your tests;
- (B) amplify your tests;
- (C) amplify your test configurations;
- (D) reproduce crashes.

The following picture provides an overview of a general CI/CD scenario, showing where STAMP can provide most of its value:

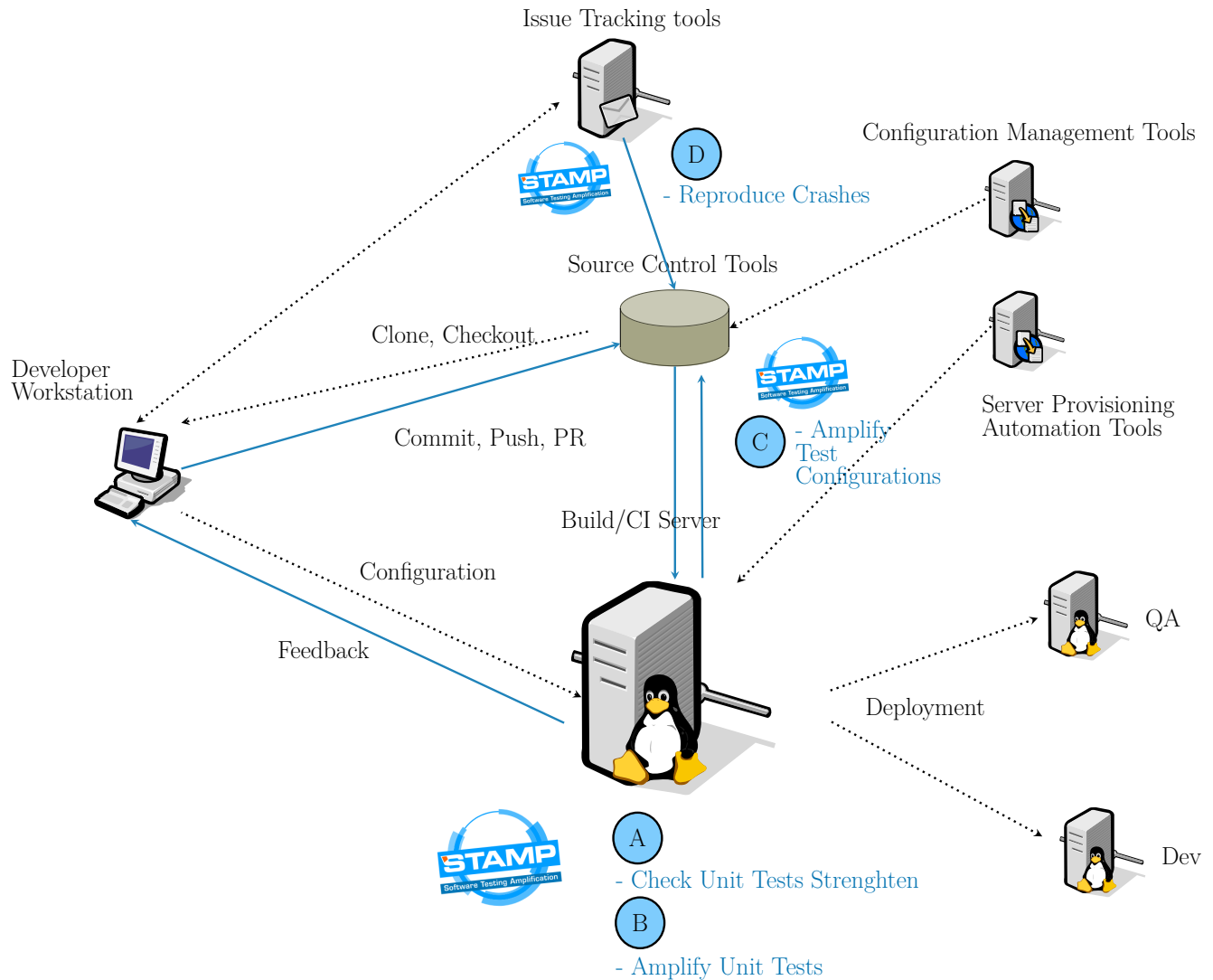


Figure 2.4: STAMP CI/CD Integration scenario.

The CI/CD Server extends its capabilities with the STAMP amplification features, which are integrated smoothly in the CI/CD cycle: unit tests are tested leveraging extreme mutation testing introduced by Descartes, unit test are amplified by the means of DSpot, test configuration are amplified, by the means of CAMP. For bugs which escape from the CI/CD pipeline and make systems crash in production, or even during test and QA stages, the related logs containing the stack-trace are analyzed with the help of Botsing and new test cases are generated to replicate the crashes. In following sections we will describe in detail the STAMP CI/CD scenario.

2.5.1 Descartes: test your tests within CI/CD

As per the official description available on STAMP public repository, “Descartes evaluates the capability of existing test suites to detect bugs using extreme mutation testing”. This means that Descartes doesn’t produces new code but rather offers an effective way to assess existing

test cases strength. This capability is exploited in the reference CI/CD scenario, whenever new code is pushed back in the GitHub repository. When new code is pushed on the repository, Descartes runs against test cases related to code changes or even against new test cases provided along with the push operation. Results are made available to the developer with reports that show the number of pseudo tested methods as well as the mutation coverage reached by the assessed test suites.

Following figure shows an example:

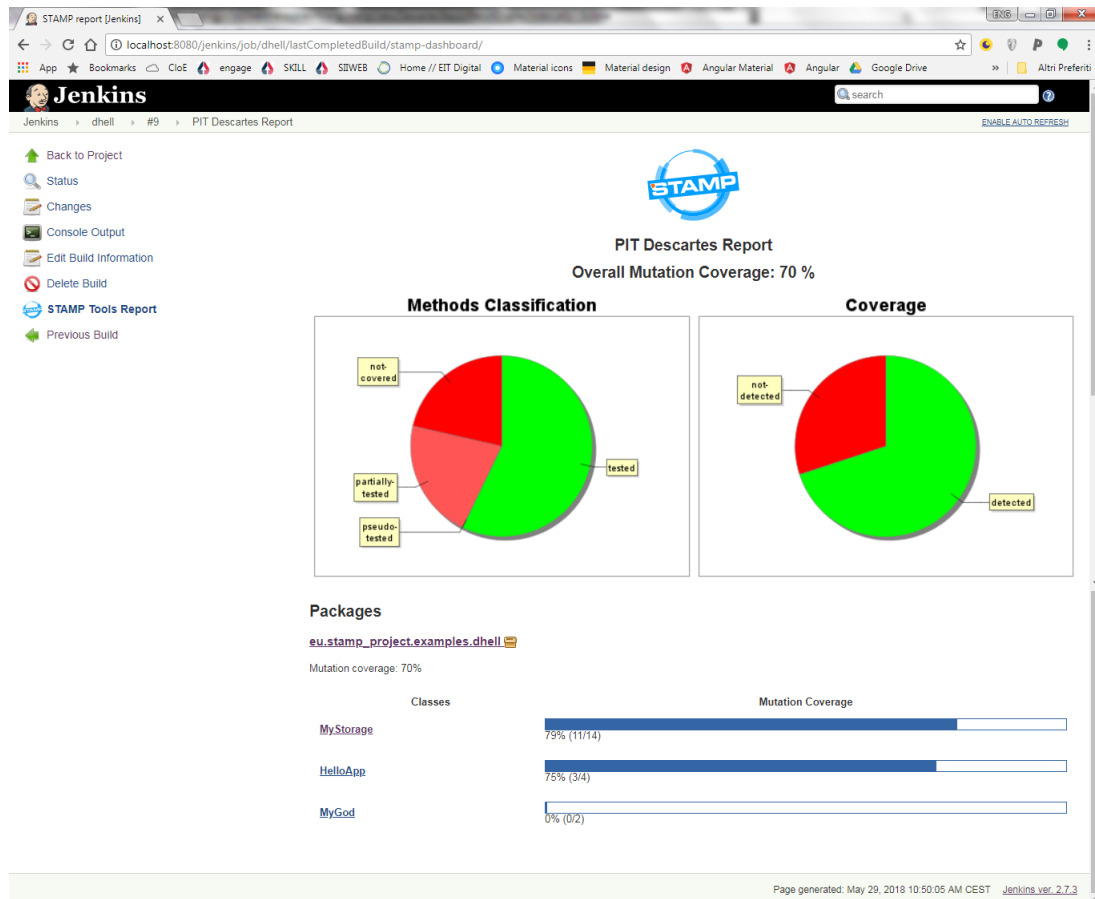


Figure 2.5: Detailed report about Descartes execution in a Jenkins build

This report shows the distribution of methods Descartes classification for the current build. Developers can check the existence of pseudo-tested methods (methods covered by test suite, but no test case fails when their method bodies are removed, i.e. when extreme mutation testing is applied). It's possible to check also the trend of mutation outcomes along with several builds and evolution of the code base:

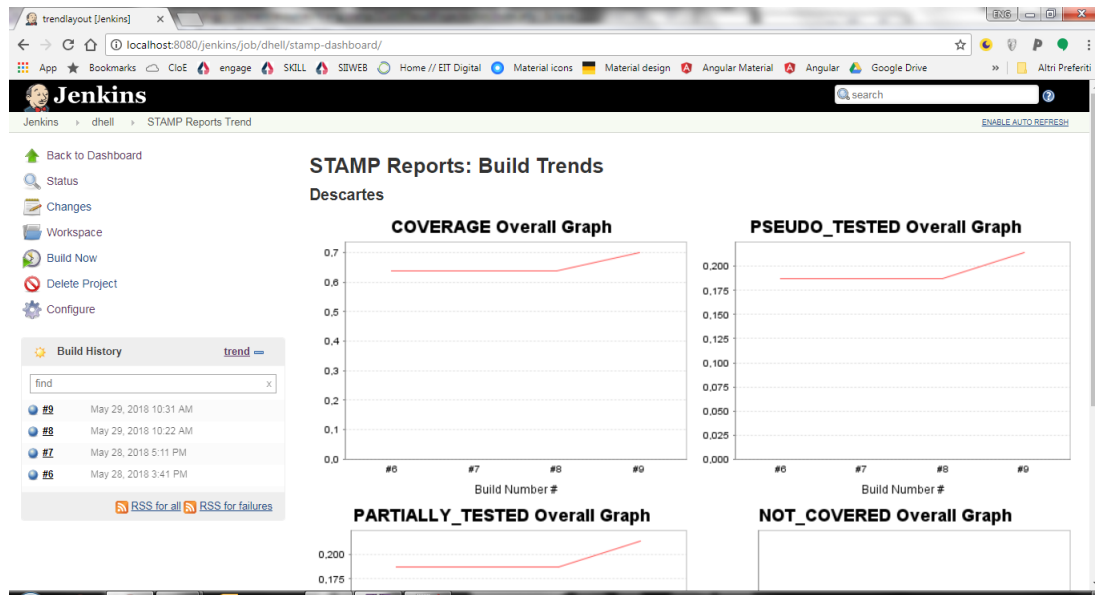


Figure 2.6: Descartes assessment over the time

Future work

Currently the plugin supports freestyle jobs, but it will be extended to support by default Jenkins pipelines. We will optimize the execution adding capabilities to check only test cases related to code changes.

2.5.2 DSpot: amplify your tests within CI/CD

When new code is pushed on the repository, DSpot can run against test cases related to code changes or even against new test cases provided along with the push operation. If new test cases are generated by DSpot, a new branch containing them and a related Pull Request is opened on the GitHub repository, letting developers to choose whether adding them to the code base or not. This scenario fits well in a typical GitFlow branching model, where code pushed back in the develop branch can trigger the execution of DSpot to enhance test cases, which are then pushed back in the master branch or in another release branch. The following figure shows how to adopt a branching model which supports the test amplification stage:

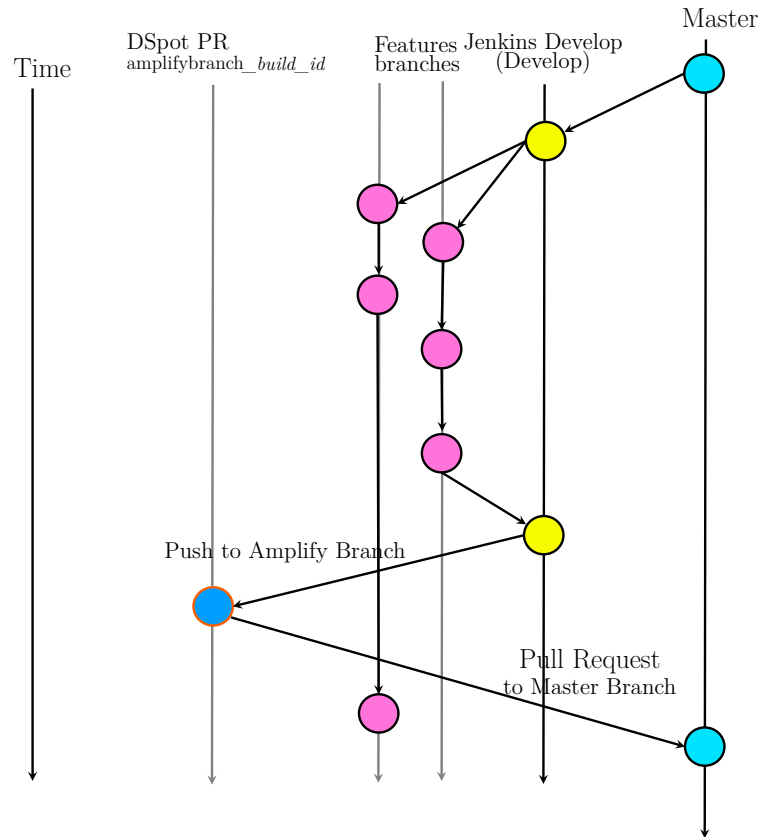


Figure 2.7: Branching model with explicit branching to support test amplification.

In this branching model the parallel development proceeds as in the ordinary GitFlow: each developer opens a feature branch to work on, and when he's happy with his own changes, he pushes back to the develop branch, which acts as an integration branch. This branch is named "Jenkins branch (develop)" just to make it explicit that in this branch Jenkins will apply test amplification on new code. The code pushed back from the feature branch to develop branch triggers DSpot execution on Jenkins: a special pipeline executes DSpot, create a new branch on which pushes new test cases generated by DSpot and then opens a pull request from this new branch back to the master branch. The developer responsible to merge the develop branch to the master one can then decide whether to accept or not the new test cases bound to the pull request. In the following, we discuss the structure of the pipeline needed to get the test amplification integrated in the CI/CD scenario. Pipelines are Jenkins jobs enabled by the Pipeline plugin and built with text scripts that use a Pipeline DSL (Domain Specific Language) in Groovy programming language. Usually a pipeline is comprised of several steps, which are the building blocks of a CI/CD workflow, and stages, made of one or more steps configured to perform high level workflow steps (for a more general introduction on Jenkins pipelines, check the official Jenkins documentation at <https://jenkins.io/doc/book/pipeline/getting-started/>). Jenkins pipeline execution can be triggered by specific event or scheduled according to specific rules. For instance, a pipeline can be triggered when a push event is notified by the source code repository: this is the kind of event we leverage to integrate test amplification within the CI/CD scenario. The following diagram shows the pipeline structure containing the test amplification stage:

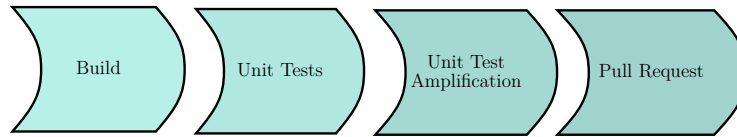


Figure 2.8: Jenkins pipeline containing a test amplification stage.

After an initial configuration step, Jenkins checks the code out, builds it, executes unit tests and then triggers DSpot execution to amplify existing test cases. If new tests are generated, a new branch containing them is created, and a pull request is created, in order to let developers decide whether accept changes coming from the develop branch with the additional amplified test cases, or not. It's important to note that test amplification should happen only when new code is pushed back in the develop branch: this means that the pipeline code needs some logic to check which is the current branch. Complete pipeline script is available on the official GitHub repository in DHell demo project at <https://github.com/STAMP-project/dhell/blob/master/Jenkinsfile>, and we configured it in the STAMP Jenkins instance, installed within a VM belonging to STAMP collaborative platform - see <http://vmi2.stamp-project.eu/jenkins/>). In the following section relevant parts of the pipeline are highlighted and discussed:

```
1 pipeline {
2   ...
3   stages {
4     stage('Build') {...}
5     stage('Unit Tests') {...}
6     stage('Amplify') {
7       when {branch 'jenkins_develop'}
8       steps {
9         withMaven(maven: 'maven3') {
10          sh 'mvn eu.stamp-project:dspot-maven:amplify-unit-tests -Dpath-
11            to-properties=dhell.dspot -Damplifiers=TestDataMutator -
12            Dtest-criterion=JacocoCoverageSelector -Diteration=1'
13          }
14          sh 'cp -rf target/dspot/output/eu src/test/java/'
15        }
16      stage('Pull Request') {
17        when {branch 'jenkins_develop_branch'}
18        steps {
19          sh 'git checkout -b amplifybranch-${BUILD_NUMBER}'
20          sh 'git commit -a -m "added amplified test cases"'
21          withCredentials([usernamePassword(credentialsId: 'github-user-password',
22            passwordVariable: 'GIT_PASSWORD', usernameVariable: 'GIT_USERNAME')
23          ]) {
24            sh('git push https://${GIT_USERNAME}:${GIT_PASSWORD}@github.com/
25              STAMP-project/dhell newbranch${BUILD_NUMBER}')
26          }
27          sh 'hub pull-request -m "develop branch with amplified test cases"
28            ,
29          }
30        }
31      }
32    }
33  }
```

```

27     }
    ...
29 }

```

1. The Amplify stage is an optional stage, executed only when the pipeline execution is triggered by a push event on the Develop branch. To get this, the condition at line 7 is evaluated first. If true, then:
 - 1.1. DSpot is executed as a Maven goal at line 7, and
 - 1.2. new test cases are copied from the DSpot default folder `target/dspot/eu` to the Maven standard test directory `src/test/java` (line 13);
2. the Pull Request stage is optional too, and is executed when the current branch is the Develop one (line 17), as for Amplify stage:
 - 2.1. the new branch is created (line 19);
 - 2.2. new test cases are committed (line 20);
 - 2.3. new test cases are pushed back to the amplification branch (line 22);
 - 2.4. a pull request to the master branch is opened (line 24);

The following picture shows the pipeline execution triggered by a push event into the `jenkins_develop` branch:

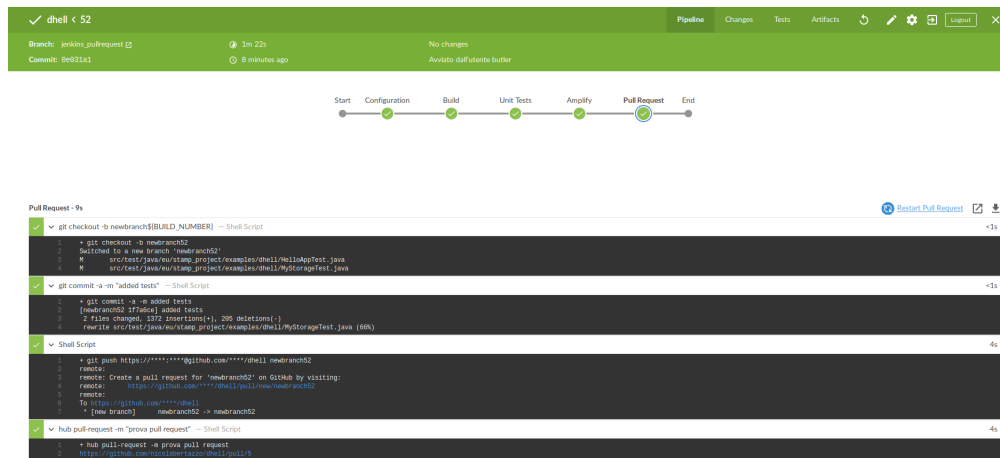


Figure 2.9: Pipeline with test amplification execution in the Develop branch

In the Jenkins pipeline execution console it is possible to see the execution of the Pull Request steps:

```

Pull Request - 9s

✓ git checkout -b newbranch${BUILD_NUMBER} -- Shell Script
1 + git checkout -b newbranch52
2 Switched to a new branch 'newbranch52'
3 M   src/test/java/eu/stamp_project/examples/dhell/HelloAppTest.java
4 M   src/test/java/eu/stamp_project/examples/dhell/MyStorageTest.java

✓ git commit -a -m "added tests" -- Shell Script
1 + git commit -a -m added tests
2 [newbranch52 1f7a6ce] added tests
3 2 files changed, 1372 insertions(+), 205 deletions(-)
4 rewrite src/test/java/eu/stamp_project/examples/dhell/MyStorageTest.java (66%)

✓ Shell Script
1 + git push https://****:****@github.com:****/dhell newbranch52
2 remote:
3 remote: Create a pull request for 'newbranch52' on GitHub by visiting:
4 remote:   https://github.com/****/dhell/pull/new/newbranch52
5 remote:
6 To https://github.com/****/dhell
7 * [new branch]      newbranch52 -> newbranch52

✓ hub pull-request -m "prova pull request" -- Shell Script
1 + hub pull-request -m prova pull request
2 https://github.com/nicolabertazzo/dhell/pull/5

```

Figure 2.10: Detailed steps for the Pull Request stage

Following picture shows instead the execution of the pipeline against the branch `amplifybranch-build_number` where amplified test cases have been pushed by Jenkins thanks to the same pipeline itself, showing that in this case the Amplify and Pull Request stages are skipped:

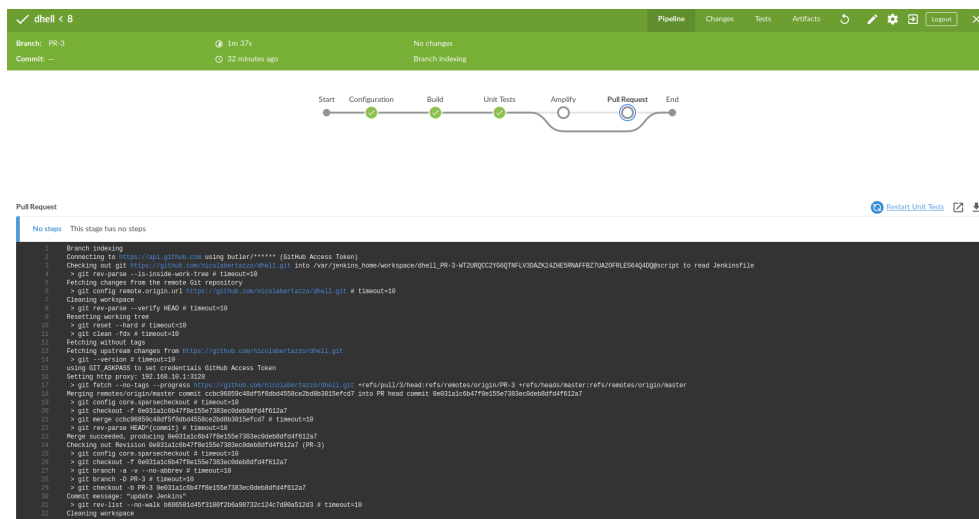


Figure 2.11: Pipeline that skips test amplification execution outside the "test amplification" branch.

As we explained previously, the pipeline execution is triggered by every “push” event in the source code repository, no matter which branch is affected by the push, so to avoid infinite loops “push new code in a branch → amplify test cases → push new test cases in a another new branch → amplify them in this new branch → etc”, it becomes clear the reason of having conditional steps in the pipeline, that trigger test amplification just in the develop branch. Now let’s see what happened on the GitHub repository. In the following picture we can see the pull

D4.3 Second public version of API and implementation of services and courseware

request opened on the master branch from the branch **newbranch43** (43 is the build number):

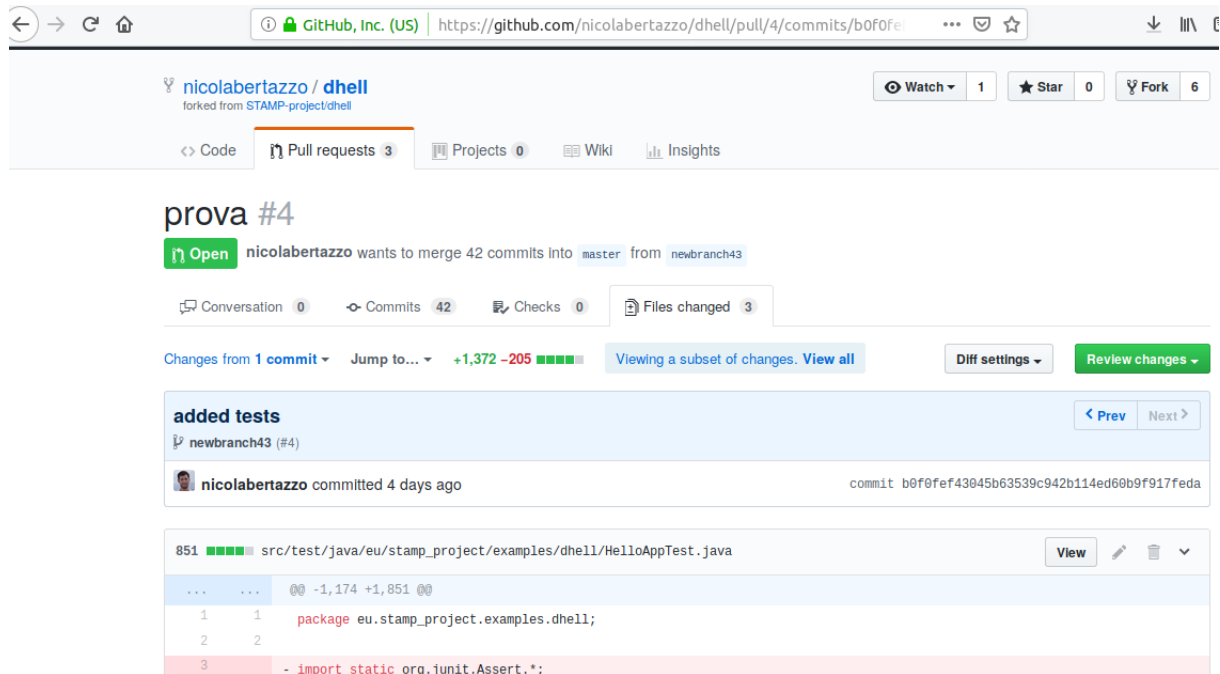


Figure 2.12: Pull Request details opened automatically by Jenkins and containing amplified test cases.

Inspecting the changes brought by the PR, we can see how existing test cases have been amplified by DSpot (in red you can see the original test case code, while in green you can see how the test case has been amplified, with new assertions and inputs):

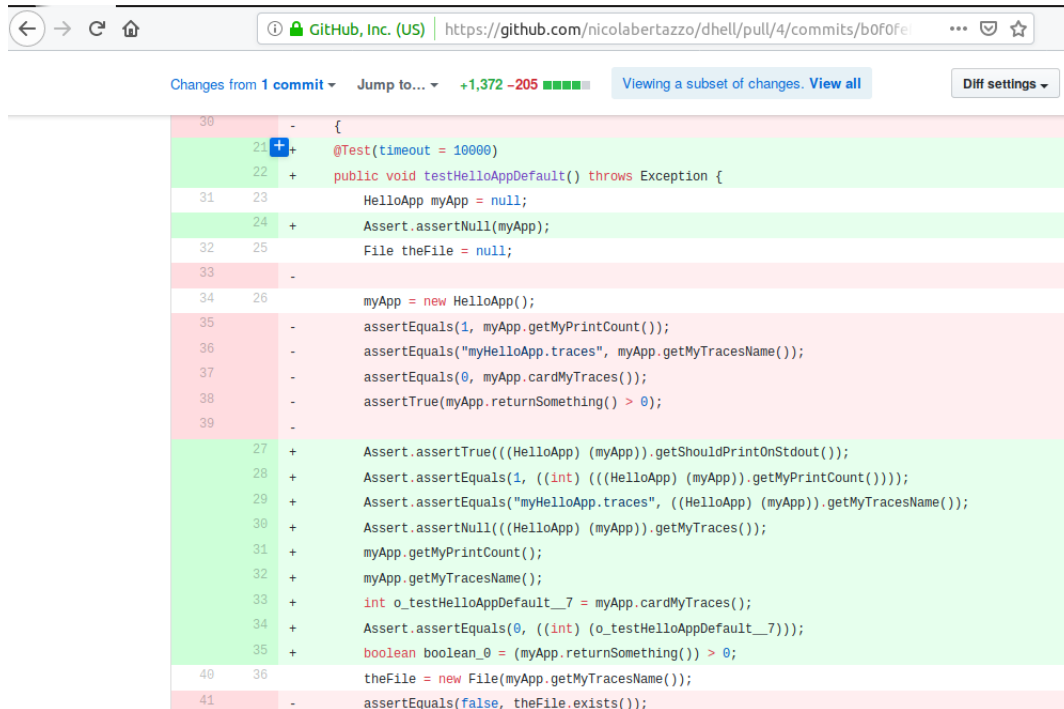


Figure 2.13: Pull Request content: test cases has been amplified with several new assertions.

Future work

We will optimize the execution adding the capability to amplify only new test cases, or test cases bound to code changes.

2.5.3 CAMP: amplify your test configurations within CI/CD

CAMP usage is related to test configuration management and to test environments provisioning. Thanks to container technology, it is possible to manage test environments with simple text files, which, in turn, can be managed within source code repositories. This makes it possible to integrate STAMP test configuration amplification capabilities in the Gitflow branching model. The approach is similar to the one we presented in the previous section: when the code coming from a feature branch is pushed back to the develop branch, a pipeline is executed with a conditional test amplification stage:



Figure 2.14: Jenkins pipeline containing a test configuration amplification stage.

The main difference relies on the fact that CAMP needs to be executed when test configurations change. This means that the pipeline code inspects the change-set related to the push event and execute CAMP only if:

- CAMP model file has been modified;
- Docker compose and Docker file have been modified.

The process is quite similar to the one described for unit test amplification, Test configuration Amplification stage will perform following steps:

1. CAMP generate: CAMP generates all new possible configurations (a YAML file for each configuration that indicates how components are wired and their configurations), starting from the CAMP model file (it can possibly generate a smaller subset of configurations which take into account all possible components);
2. CAMP realize: CAMP transforms these new configurations in actual Docker files.

Then, as seen also for unit test amplification, the Pull Request stage will open a a PR containing the new test configurations.

Future work

We will implement the pipeline, triggering the execution only when test configurations will be affected by changes. The pipeline will open a pull request with new configurations in the same way we have seen for DSpot.

2.5.4 Botsing: reproduce automatically production crashes within CI/CD

Botsing supports developers to speed up the bug fixing process, reproducing automatically an error with the generation of a test case able to replicate it. In the CI/CD context, unexpected bugs usually arise in test and QA environment or, sometime, in production systems. In these contexts the ordinary process of bug management requires to submit the bug in an issue tracker. When the bug is opened and assigned to a developer, the bug fixing process starts. Following diagram summarizes the process:

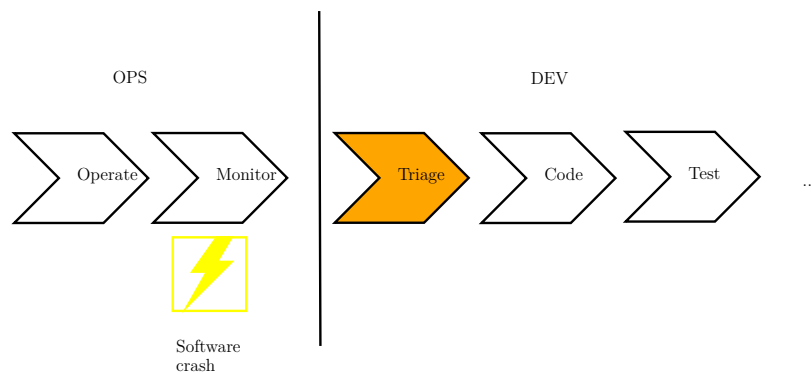


Figure 2.15: A crash occurring during operations triggers the bug fixing process within development.

Monitoring detects the crash, all relevant bug information are collected and sent to development that starts with the bug triage (to evaluate, prioritize and assign the resolution of the defect), and then the bug fixing can start within the ordinary stages of a standard Dev phase. Considering the GitFlow branching model, a special branch is created for hot fixes:

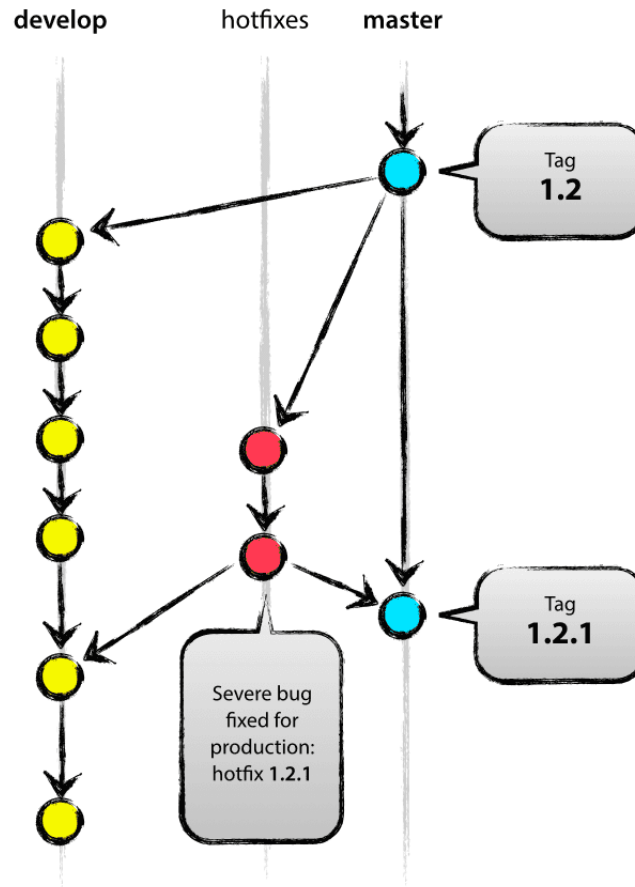


Figure 2.16: Gitflow branching model to manage hot-fixes
<https://nvie.com/posts/a-successful-git-branching-model/>

The defect triage is essentially a human activity, but, thanks to STAMP, a step forward a more automation in the triage phase relies on the integration of the issue tracker in the overall CI/CD scenario, embedding the Botsing features directly in the issue tracker. One of the most used issue trackers in the software development world is Atlassian Jira: it has a flexible configuration model to support companies development and QA processes, but the most interesting feature for our purpose is its very rich programming model. Thanks to the Atlassian SDK (<https://developer.atlassian.com/server/framework/atlassian-sdk/>), it is possible to develop Jira extensions, in form of plugin, making it possible to integrate Jira with external tools and systems. An interesting feature of Jira is its capability to fire “events” (issue creation, issue modification, etc) for which one or more “listeners” can register themselves, in order to perform actions when a new event is triggered. It is possible to develop custom listeners able to execute custom code corresponding to a given event. Here comes the Botsing integration: a specific listener listens for “bug creation” event, check whether the minimum information to run Botsing are available, and trigger the execution of Botsing itself. If the execution goes well, a pull request is sent to GitHub, in the Hot-fix branch, along with the generated test case. The following schema summarizes this scenario:

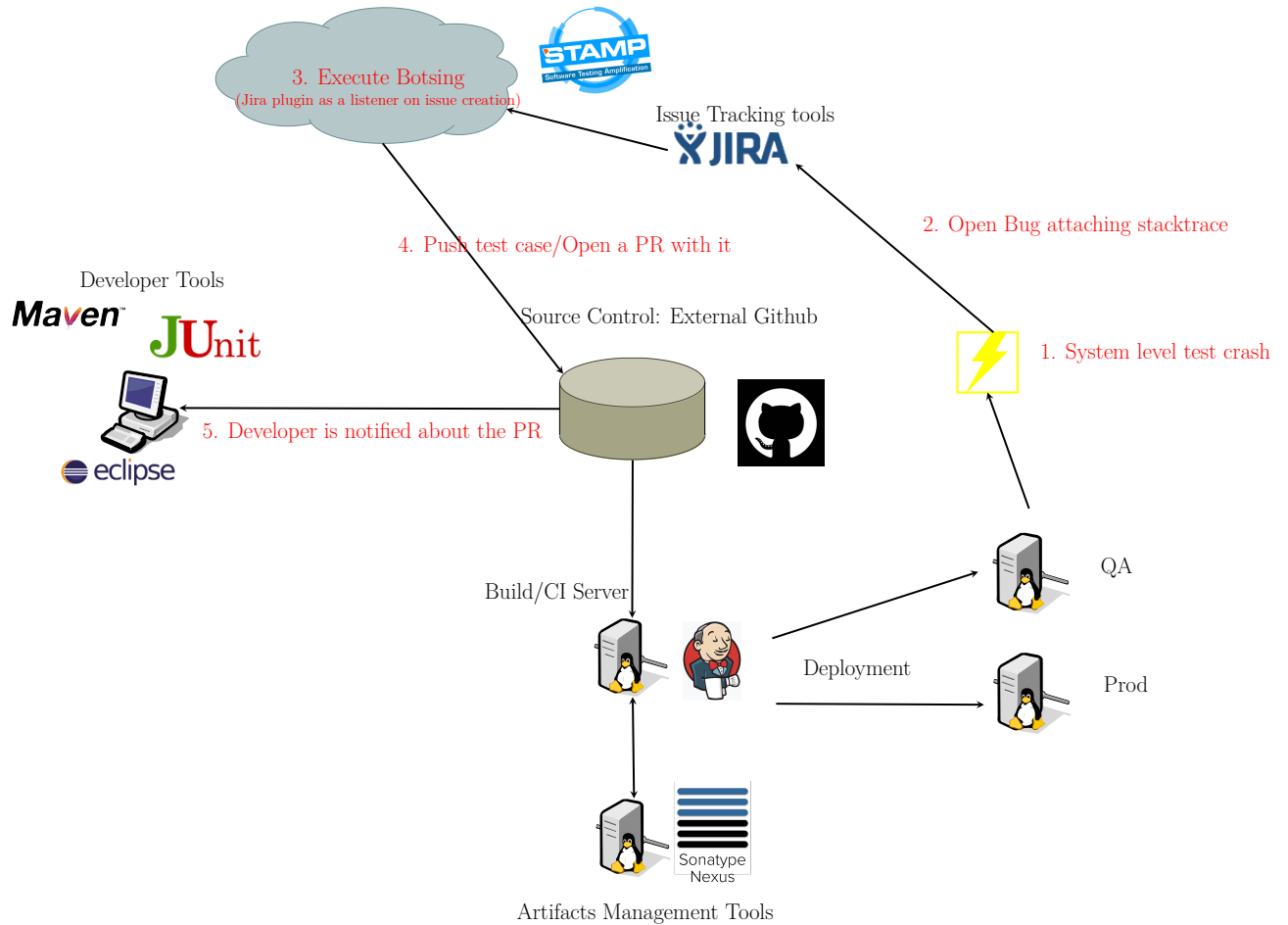


Figure 2.17: Bug triage automation Botsing flow within the CI/CD scenario.

Following a description of the steps involved:

1. A crash happens and the log containing the stack-trace is collected from operations;
2. a new bug is filled to the Jira instance, along with the log file containing the stack-trace and other needed information (software version, etc).
3. The Jira event “new bug created” triggers Jira Botsing plugin execution, which generates a new test case able to reproduce the crash;
4. then the Jira Botsing plugin opens a pull request on GitHub, containing the new test case.
5. The developer is then notified about the PR from GitHub and the defect resolution process starts.

Jira listeners can be configured to be very smart, and to act not just in the case of a bare “issue creation” event: more complex conditions can be configured and whenever they have been met, the Botsing execution can run (for instance: execute Botsing only if log file has been attached, the application version is available, etc).

Future work

We will implement the Jira plugin needed to automate the crash reproduction. This plugin will then open a Pull Request to GitHub repository containing the generated test case.

Chapter 3

STAMP ecosystem: state of the art

Around the 4 core tools developed in STAMP, we also built an ecosystem of plugins and integration to support various technological development contexts and scenarios. We classified the ecosystem in the following components:

1. build systems: software tools designed to automate the process of building software systems, starting from source code. In addition to the simple creation of software executable, modern build systems can accomplish a variety of other goals, related to testing, deployments, documentation generation and more.
2. Developer productivity tools: tools that support software development process each developer is involved in. While the bare minimum set of tools needed by a developer is an editor to write the code, and a compiler to build it, software development needs several activities to ensure the proper software quality; developers usually uses IDE (Integrated Development Environment) which integrate different facilities the developer needs, to write software solutions that work: source code editors, build automation tools, debuggers, dependency managers, etc.
3. source code repositories: the main purpose of a repository is to store the source code, but modern source code repositories offer the possibility to developer communities to collaborate on the same code base, make reviews, share ideas, proceed with parallel development. Modern source code repositories usually offers API to integrate external systems, making them listeners to events (commits, push actions, etc) which in turn trigger actions to analyze, build, test, deploy the source code;
4. Configuration and Provisioning Systems: systems that streamline service instance lifecycle processes. Developers and operations need these systems to test and operate the software. Modern configuration and provisioning systems leverage key concepts as "Infrastructure as code" (target systems described by text files containing specific source code - see <https://martinfowler.com/bliki/InfrastructureAsCode.html>) and container technology (a method to package a software system so it can run, with its dependencies, isolated from other processes. Containers make easy infrastructure as code (see <https://rancher.com/containers-making-infrastructure-code-easier/>).
5. Continuous Integration/Continuous Delivery systems: it is very rare the software solutions are built by a single developer. CI/CD systems orchestrate the interaction among several systems (source code repositories, artifacts repositories, configuration management and provisioning systems, static analysis systems and so on) in order to built software solution in a reliable, repeatable way. CI/CD systems can also listen for events coming from source code repositories, to trigger software builds.

6. STAMP tooling API: API, connectors, services to integrate or embed STAMP in external systems;
7. STAMP cookbooks, documentation and courseware.

The following picture shows an overview of the ecosystem:

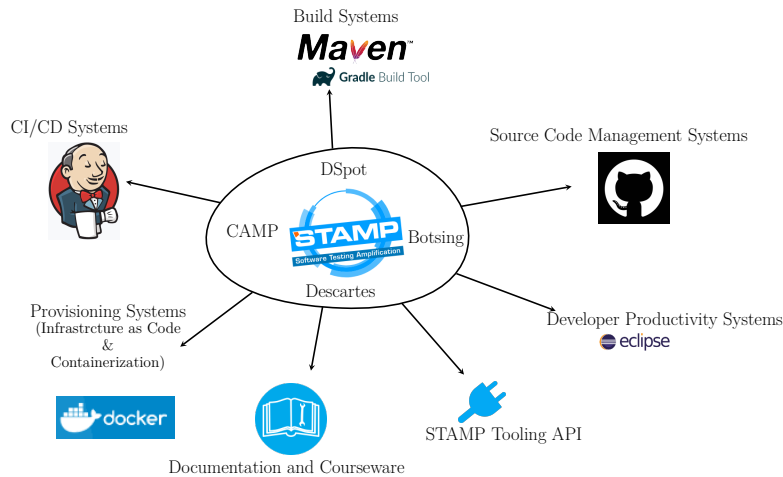


Figure 3.1: STAMP ecosystem

In following sections we will describe in more detail what we developed within the ecosystem.

3.1 DSpot

3.1.1 Build Systems

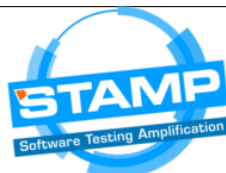
DSpot Maven plugin

One of the first integration that have been made for DSpot was a Maven plugin to let developers invoke DSpot as a special Maven goal. This plug-in exposes the `amplify-unit-tests` goal that lets a developer to run DSpot amplification within the Maven build process. The usage is very simple: from the Maven Java project root folder (which contains the `pom.xml` file) the developer just run the following command: `mvn dspot:amplify-unit-tests` DSpot Maven plugin is available on official Maven repositories (<https://mvnrepository.com/artifact/eu.stamp-project>), so it is not necessary to configure a special Maven repository in the POM file of the target project. DSpot parameters are passed as Maven parameters, while specific properties related to the target project are passed in a properties file. DSpot Maven plugin code, along with the documentation, is available at <https://github.com/STAMP-project/dspot>

3.1.2 Developer Productivity Tools

STAMP IDE: DSpot plugin

Eclipse has been elected as the reference IDE for STAMP. It's a full-featured, open source IDE, with a powerful plugin mechanism that simplifies the extension of the IDE with new features. In addition, Eclipse community is one of the largest software developers' communities. Indeed, having STAMP functionalities available as Eclipse plugins is one of the more promising ways to promote its usage. DSpot, Descartes and Botsing are equally supported, with configuration



wizards, the possibility to save different configurations and to check the execution within the common Eclipse output console and dedicated output views. Getting the STAMP IDE setup is as simple as configuring a new Eclipse repository, as described in <https://github.com/STAMP-project/stamp-ide>:

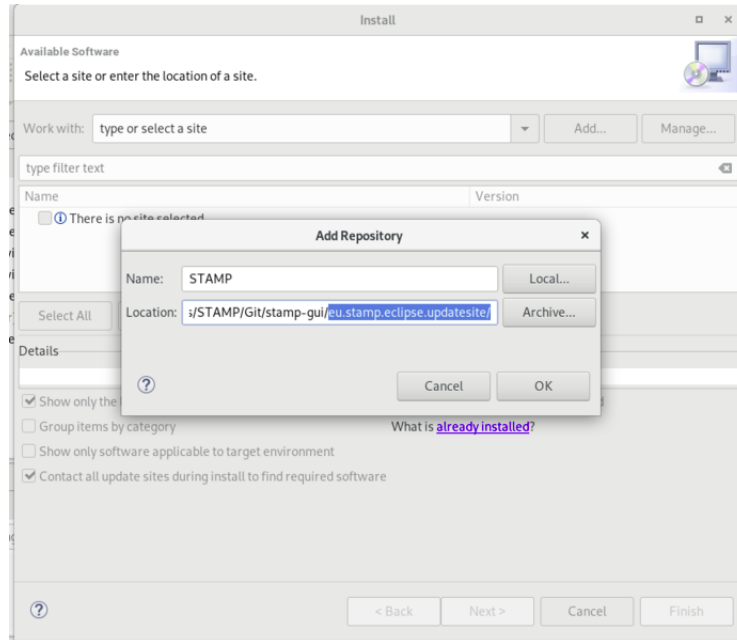


Figure 3.2: STAMP IDE repository configuration

And selecting the STAMP plugins to be installed from the repository list

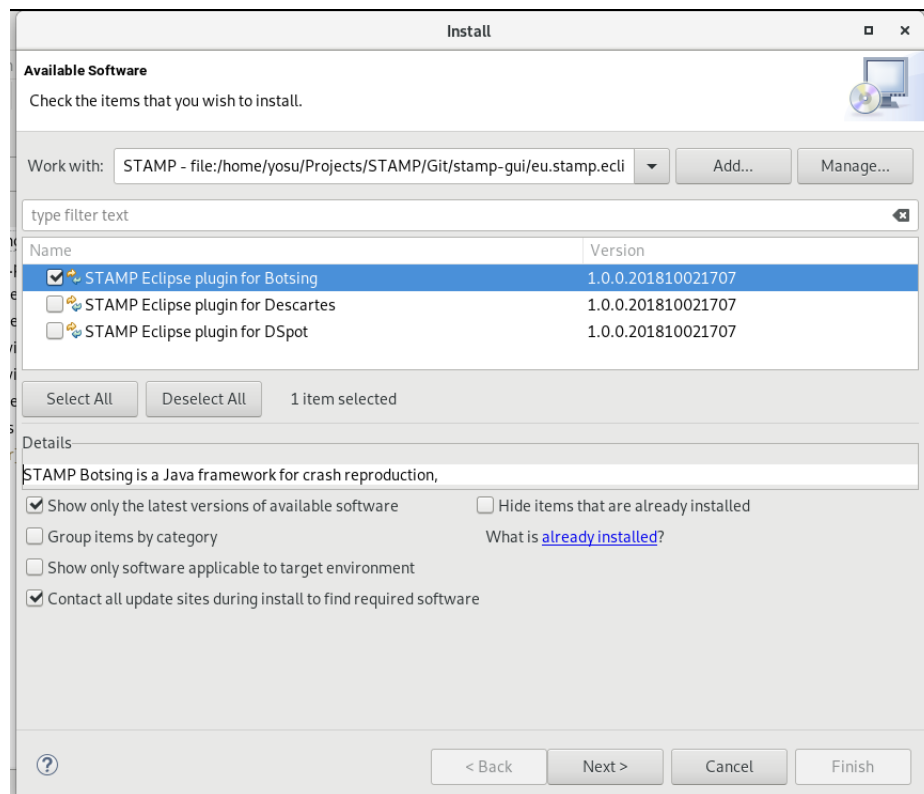


Figure 3.3: STAMP IDE Plugins installation

A STAMP menu lets STAMP users to access to the wizards for the different STAMP core tools on selected compatible projects:

D4.3 Second public version of API and implementation of services and courseware

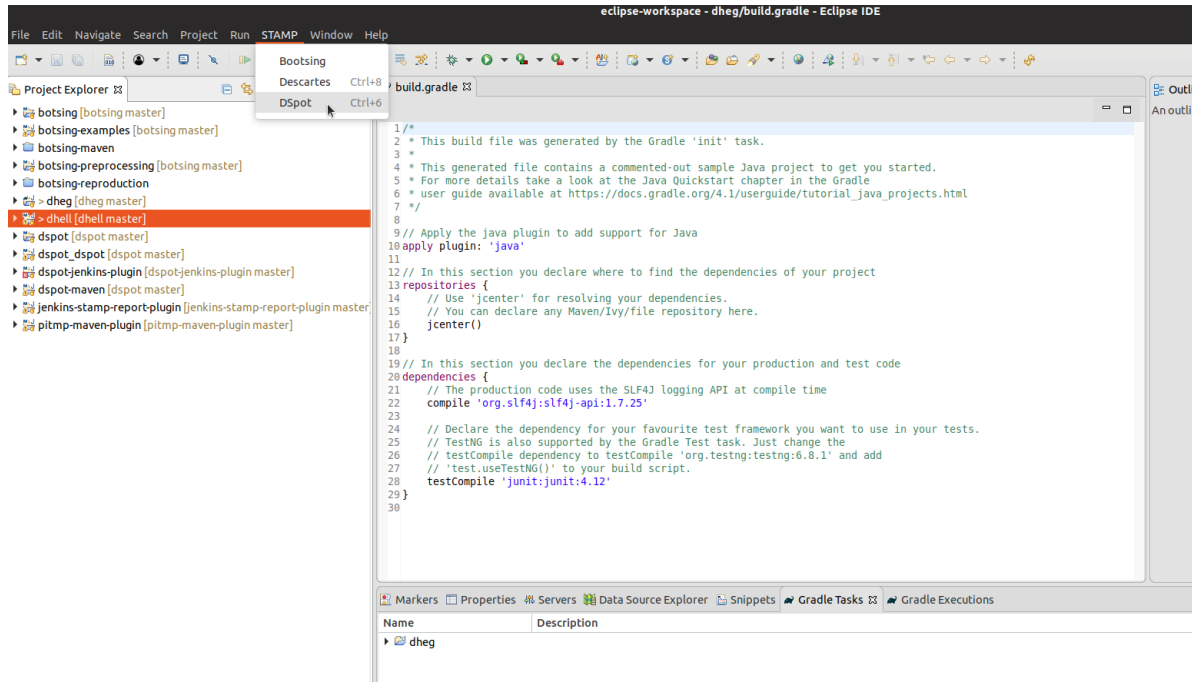


Figure 3.4: STAMP IDE menu

DSpot wizard lets developers to configure all DSpot parameters for both the target project (i.e. dspot.properties) and its execution, save it as an Eclipse run configuration, and launch it against the selected project test cases:

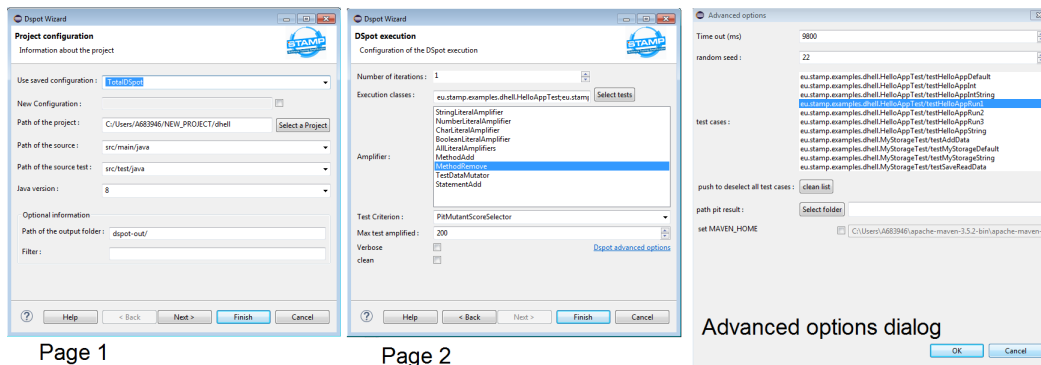
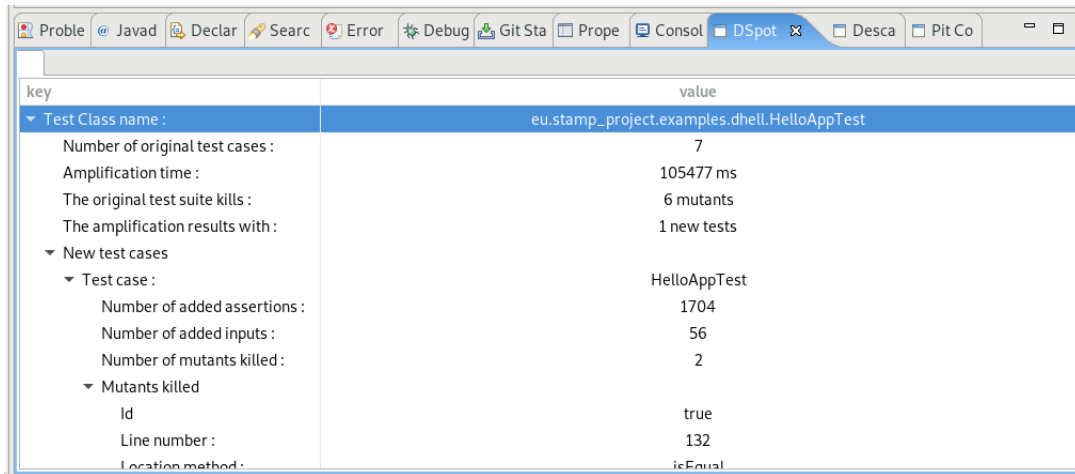


Figure 3.5: STAMP IDE: DSpot configuration wizard

Upon execution completion, DSpot results are visualized in a dedicated view, and the amplified tests placed in the selected output folder.



key	value
Test Class name :	eu.stamp_project.examples.dhell.HelloAppTest
Number of original test cases :	7
Amplification time :	105477 ms
The original test suite kills :	6 mutants
The amplification results with :	1 new tests
New test cases	
Test case :	HelloAppTest
Number of added assertions :	1704
Number of added inputs :	56
Number of mutants killed :	2
Mutants killed	
Id	true
Line number :	132
Location method :	isEqual

Figure 3.6: STAMP IDE: DSpot results view

Eclipse plugin for DSpot documentation is available at: https://github.com/STAMP-project/stamp-ide/blob/master/README_DSpot.md. DSpot plugin for Eclipse requires the selection of a Maven-managed target project in the Eclipse project explorer.

Additional features have been implemented for the DSpot plugin for Eclipse, namely:

- Support for DSpot v1.2.2
- Manage of DSpot Eclipse runtime configurations. They can be executed from the DSpot Wizard or from the Eclipse run configuration facility.
- Wizard form validation: all form fields are checked for valid (and mandatory) values.
- Wizard form tooltips: all form fields include explanatory tooltips that popup when the mouse pointer hovers on their labels
- Complete configuration of target project properties (i.e. dspot.properties)
- Complete configuration of DSpot execution parameters
- Upon DSpot execution failure, users are notified and referred to the DSpot (<https://github.com/STAMP-project/dspot/issues>) or the STAMP IDE(<https://github.com/STAMP-project/stamp-ide/issues>) issue tracker, depending which one caused the issue, so they can report it.

3.1.3 CI/CD Systems

DSpot Jenkins plugin

DSpot Jenkins plugin lets developers to setup a freestyle job or a pipeline to execute a test amplification stage. All parameters needed by DSpot can be provided in the Jenkins graphical user interface:

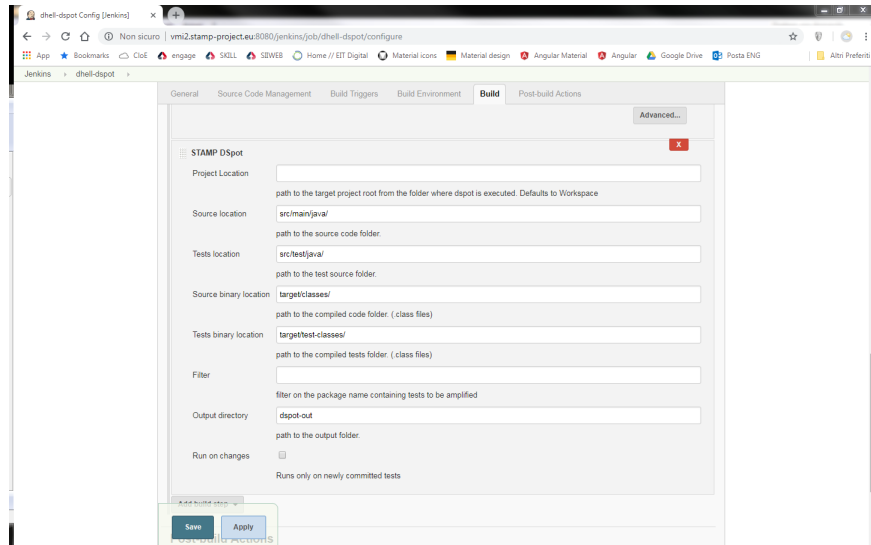


Figure 3.7: Jenkins: configuring a freestyle job to run DSpot

At the end of the execution, a detailed execution report, along with the new generated test cases are available within the build workspace. Source code and more information are available at <https://github.com/STAMP-project/dspot-jenkins-plugin>.

3.1.4 STAMP Tooling API

DSpot API

STAMP ecosystem has several tools that let to use DSpot within IDE, CI/CD servers, build systems. While these tools cover the majority of developer needs, it is also possible to embed it in other tools or custom software packages. The official DSpot documentation (<https://github.com/STAMP-project/dspot#using-dspot-as-an-api>) shows how to proceed programmatically setting up and feeding DSpot with test cases to amplify.

3.1.5 Documentation

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

1. Maven project
2. Gradle project

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

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

We setup a new virtual machine containing a clean environment, configured to run DSpot within the DHell project. The VM is available at <https://release.ow2.org/stamp/VMs/18.10/>

3.2 Descartes

3.2.1 Build Systems

Maven support

Descartes is a PIT plugin and this let Descartes piggybacks PIT Maven plugin. In this way it is possible to use Descartes simply configuring it as a PIT mutator, adding Descartes as a dependency in the POM file. Running Descartes against own test suites is as simple as typing: `mvn org.pitest:pitest-maven:mutationCoverage -DoutputFormats=HTML` As for the other STAMP tools, even Descartes is available in Maven Central repository, so Descartes end users don't need to configure external third party Maven repositories. Descartes source code along with official documentation is available at <https://github.com/STAMP-project/pitest-descartes>

PitMP - Maven plugin to execute Descartes against Maven multi-module projects

PIT has a limitation: it can't be run on Java multi-module projects. And this lead also to the consequence that Descartes can't be run on a Java multi-module project. So we developed a Maven plugin that extends PIT (and eventually Descartes) usage also to Java multi-module projects. To apply Descartes to a multi-module project, one can use directly the Maven command line or configure PitMP within the POM file as described in the official PitMP documentation available in the repository: <https://github.com/STAMP-project/pitmp-maven-plugin>

Depending of the release you want to use of PIT, a PitMP version is needed, according to the following table:

Table 3.1: PitMP compatibility

PIT release	PitMP release	how to use PitMP
1.4.2	1.3.5, 1.3.6	Maven Central
1.4.0	1.3.4, 1.3.3, 1.3.2, 1.3.1, 1.3.0, 1.2.0	Maven Central
1.3.2	1.1.6, 1.1.5	Maven Central
1.3.1	1.1.4	Maven Central
1.2.1, 1.2.2, 1.2.4, 1.2.5, 1.3.0	not tested	Maven Central
1.2.0, 1.2.3	1.0.1	git clone & mvn install

As per other STAMP artifacts, PitMP is available in Maven Central repository.

Gradle

What we have seen for Maven support holds also for Gradle: PIT comes with a Gradle plugin and this can be leveraged to use Descartes as the mutation engine for the current Gradle project. The availability of Descartes in Maven central repositories makes it usage as simple as configuring an ordinary Gradle task in the Gradle build file, so launching a Descartes assessment is as simple as typing

```
gradle pitest
```

The pitest task is configured in the Gradle build file to use Descartes as mutation engine. More information are available in the STAMP official repository, at <https://github.com/STAMP-project/pitest-descartes#gradle>

3.2.2 Developer Productivity Tools

STAMP IDE: Descartes plugin

The STAMP IDE includes the Descartes plugin that offers similar to the wizard-based support shown before for DSpot: generation of a run configuration, where the developer can provide specific operators for each mutator. As a reference to select operators, it is possible to check the official Descartes documentation at <https://github.com/STAMP-project/pitest-descartes#specifying-operators>. Each run configuration can be stored and used later. Additional Descartes plugin configuration is available at https://github.com/STAMP-project/stamp-ide/blob/master/README_Descartes.md. Descartes plugin for Eclipse requires the selection of a Maven-managed target project in the Eclipse project explorer.

Descartes execution results are available in the Eclipse console, but also in the dedicated views that show the PIT mutation score results and the Descartes issue report.

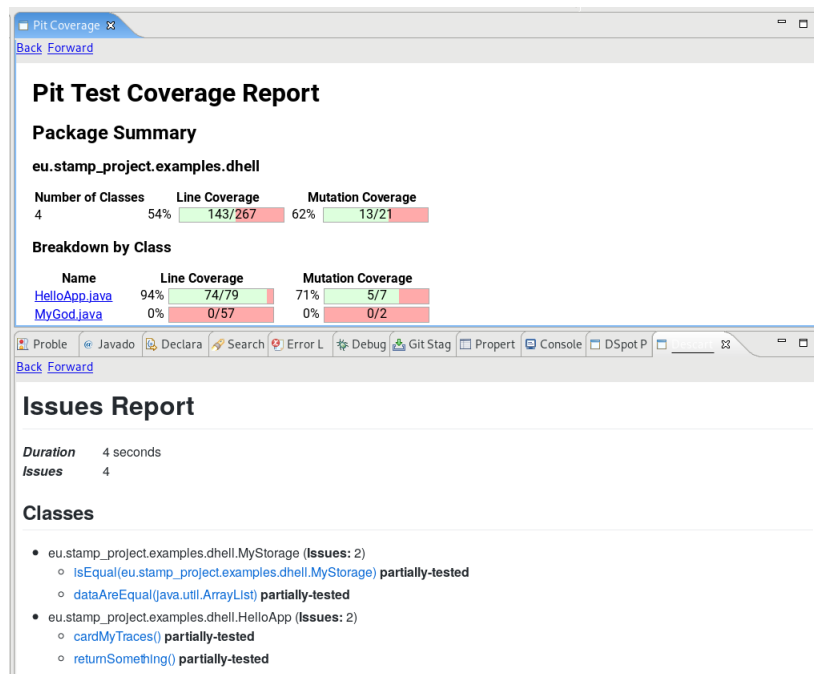


Figure 3.8: Descartes output views showing the PIT mutation score report and the Descartes issues report.

Additional features have been implemented for the Descartes plugin for Eclipse, namely:

- Support for Descartes v1.2.4
- Manage of Descartes Eclipse runtime configurations. They can be executed from the Descartes Wizard or from the Eclipse run configuration facility.
- Wizard form validation: all form fields are checked for valid (and mandatory) values.
- Wizard form tooltips: all form fields include explanatory tooltips that popup when the mouse pointer hovers on their labels
- Complete configuration of Descartes execution parameters (in Maven configuration)

- Upon Descartes execution failure, users are notified and referred to the Descartes (<https://github.com/STAMP-project/pitest-descartes/issues>) or the STAMP IDE(<https://github.com/STAMP-project/stamp-ide/issues>) issue tracker, depending which one caused the issue, so they can report it.

3.2.3 Source code repositories

Descartes GitHub App

The way to integrate GitHub repository with external services is creating GitHub Apps, which is “the officially recommended way to integrate GitHub”: <https://developer.github.com/apps/> We have seen how to use Jenkins CI at the core of a CI/CD scenario, but of course it is possible to integrate STAMP tools also with other systems. In order to integrate a GitHub repository with Descartes analysis, we developed a GitHub App able to run Descartes when a new Pull Request is created in the repository. Source code and information to use it are available at <https://github.com/STAMP-project/Descartes-GitHub-App>. We will use this GitHub App as a reference to integrate other STAMP tools.

3.2.4 CI/CD Systems

Descartes Jenkins plugin

Descartes Jenkins plugin provide Jenkins dashboard with several reports to analyze the effects of applying extreme mutation testing to current test suites. Several reports are available which let developers to inspect the number of pseudo-tested methods, the mutation coverage, and to drill down to specific methods to get detailed information:

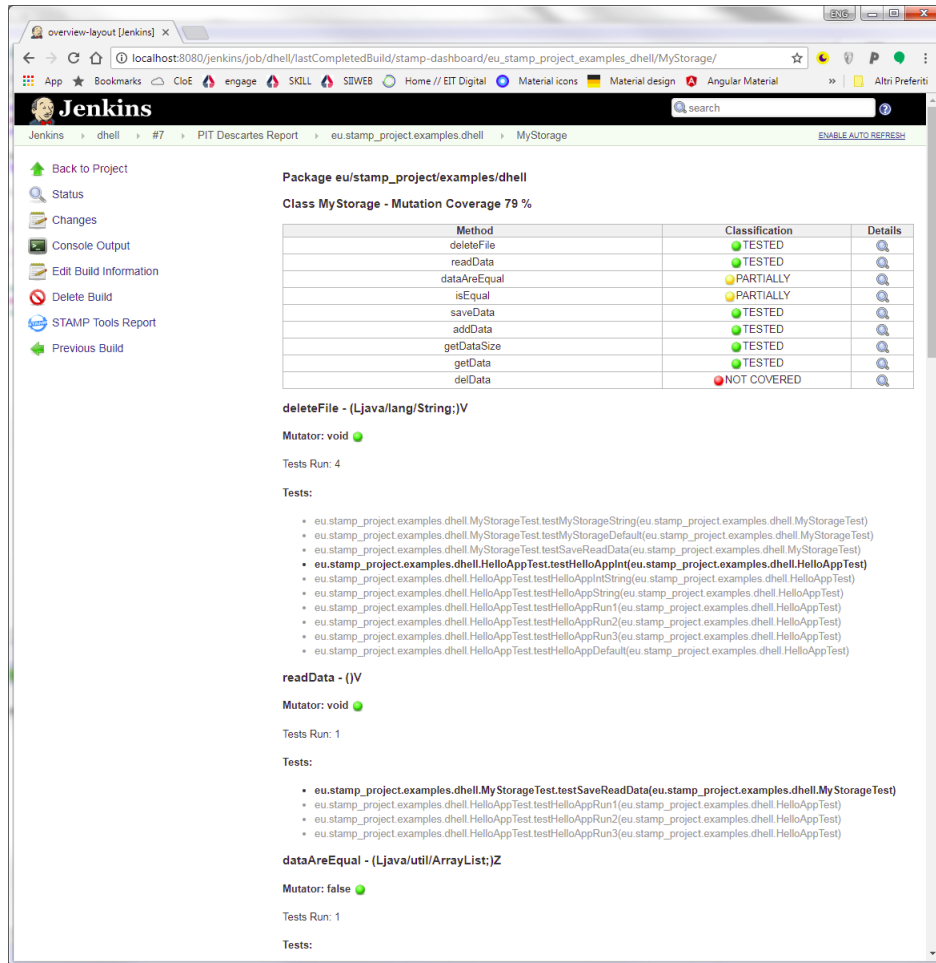


Figure 3.9: Descartes detailed analysis within Jenkins dashboard for the current build.

Along with report for the current build, also trend report are available, to check how the number of pseudo-tested methods and the mutation coverage as time goes by:

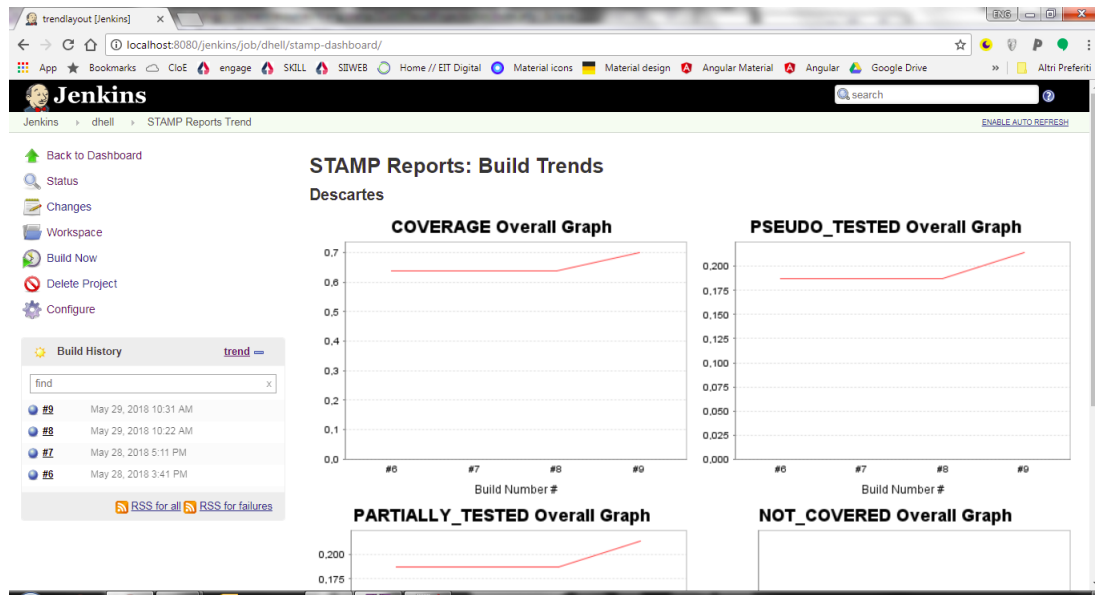


Figure 3.10: Jenkins Descartes reports: build trends.

Jenkins STAMP reports source code, along with documentation to configure and use it is available at <https://github.com/STAMP-project/jenkins-stamp-report-plugin>.

3.2.5 Documentation

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

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

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

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

The virtual machine mentioned previously is configured with a clean environment, to run also Descartes and PitMP in the DHell project. A The VM is available at <https://release.ow2.org/stamp/VMs/18.10/>

3.3 CAMP

3.3.1 Containerization

CAMP comes with ready-to-use image available in the official repository DockerHub at <https://hub.docker.com/r/fchauvel/camp/>. This image is the simplest solution to run CAMP against own test configurations, because it comes with all the CAMP dependencies packaged in the image. It exposes the three commands **generate**, **realize** and **execute** which in turn generate new possible test configurations, transform them in Docker and Docker swarm files and eventually execute them. Running CAMP as a Docker image means typing the following command:

```
$ docker run -it -v $(pwd):/camp/workspace fchauvel/camp:v1.0.0 camp generate -d workspace
```

CAMP takes as input a folder containing existing test configurations along with the CAMP configuration file (`$(pwd):/camp/workspace`). The workspace is mounted and made it available to the container thanks to the option `-v`. In the previous command the **generate** command is issued against the mounted workspace. In the similar way **realize** and **execute** commands can be launched.

3.3.2 Documentation

CAMP official documentation is available in form of GitHub pages at <https://stamp-project.github.io/camp/>. A getting started section is available at <https://stamp-project.github.io/camp/pages/setup.html>, with dedicated pages to the three CAMP commands (**generate**, **realize**, **execute**). Two case studies are available at <https://stamp-project.github.io/camp/pages/xwiki.html> and <https://stamp-project.github.io/camp/pages/citygo.html>.

3.4 Botsing

3.4.1 Build Systems

Botsing Maven plugin

A Maven plugin for Botsing have been developed in order to have Botsing features as an ordinary Maven goal. Compared to the command line usage, an important facility is given by the fact that class-path parameter is no more needed, being the dependencies given by the POM file. Source code and documentation usage is available at the official repository <https://github.com/STAMP-project/botsing>

Botsing Gradle plugin

We developed a Gradle plugin to execute Botsing as an ordinary Gradle task. Botsing required parameters need to be configured within the project Gradle build file:

- `logPath`: path to log to analyze;
- `libsPath`: dependencies classpath;

- `targetFrame`: stacktrace frame to reproduce with a automatic generated test case;

Optional parameters can be provided as well in the build file. Source code and documentation usage is available at the official repository <https://github.com/STAMP-project/botsing-gradle-plugin>

3.4.2 Developer Productivity Tools

STAMP IDE: Botsing plugin

Eclipse IDE includes a wizard-based Botsing plugin that users can use to configure the execution of Botsing and to store these configuration within the Eclipse run configuration storage for future reuse:

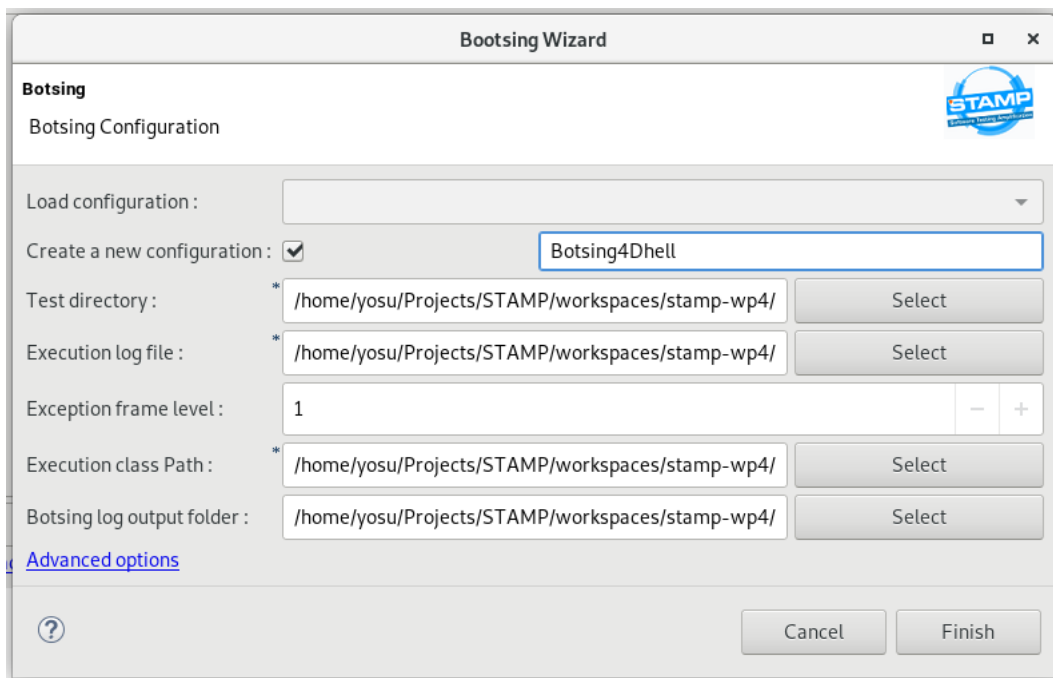


Figure 3.11: STAMP IDE: Botsing configuration wizard

Botsing execution logs are shown in the Eclipse console. They can also be store in a file selected by the user.

Additional features have been implemented for the Botsing plugin for Eclipse, namely:

- Support for Botsing v1.0.1
- Manage of Botsing Eclipse runtime configurations. They can be executed from the Botsing Wizard or from the Eclipse run configuration facility.
- Wizard form validation: all form fields are checked for valid (and mandatory) values.
- Wizard form tooltips: all form fields include explanatory tooltips that popup when the mouse pointer hovers on their labels
- Complete configuration of Botsing execution parameters

- Upon Botsing execution failure, users are notified and referred to the Botsing (<https://github.com/STAMP-project/botsing/issues>) or to the STAMP IDE(<https://github.com/STAMP-project/stamp-ide/issues>) issue tracker, depending which one caused the issue, so users can report it.

Source code and documentation for the STAMP Eclipse IDE are available at the official STAMP IDE repository at <https://github.com/STAMP-project/stamp-ide/>

3.5 Future work

The ecosystem is expected to grow, in order to provide STAMP adopters with facilities to integrate it within their toolchains. We will focus mainly on:

1. CI/CD systems: we will integrate CAMP in Jenkins, defining reference pipelines. We will consider also to develop a specific Jenkins plugin to provide CAMP users with a graphical interface to configure it within Jenkins builds.
2. source code repositories: we will implement a GitHub App to execute DSpot due to push events, in order to implement the same scenario we defined around Jenkins.
3. issue trackers: we will develop a Jira plugin to support the automatic crash reproduction to ease the bug fixing process.
4. build systems: we will enhance the Gradle support, developing plugins for Botsing and DSpot. We will consider also the possibility to have Maven and Gradle plugin for CAMP, to integrate it in the build process they support.

Along with these new developments we will keep current ecosystem components aligned with the evolution of STAMP core tools

As new components are developed and the existing ones are updated with new features and bug fixes, we will keep aligned the documentation. We will write new tutorials for Botsing and CAMP and we will setup a new virtual machine containing a packaged STAMP CI/CD environment, with a Jenkins instance already configured with demo pipelines, to use as a working reference

Several sample projects are available in the STAMP official repository to try STAMP tools: DHell (<https://github.com/STAMP-project/dhell>, a sample Java Maven project), DHeg (<https://github.com/STAMP-project/dheg>, a sample Java Gradle project) and others. Website has a new download section (<https://www.stamp-project.eu/view/main/download>) with references to source code and documentation about STAMP core tools and several STAMP ecosystem components.

Chapter 4

Conclusion

4.1 CI/CD scenario

During the upcoming last project year we will focus to setup a complete CI/CD demo in one of the environments made available in the STAMP collaborative platform: we already setup the “test your tests” scenario in the Jenkins instance available in STAMP collaborative platform, at <http://vmi2.stamp-project.eu/jenkins/>. In addition we implemented the "Amplify your tests in CI/CD" scenario, configuring the DHell project (<https://github.com/STAMP-project/dhell>) with the creation of the integration branch `jenkins_develop` described previously, and creating a Jenkins file containing the pipeline used in the GitFlow scenario (<https://github.com/STAMP-project/dhell/blob/master/Jenkinsfile>). In the next months we will enrich it with the other integration activities, in order to have a complete working reference use case, with the development of a Jira plugin for Botsing integration, the installation of CAMP in the demo environment and the configuration of separate pipelines for CAMP and Descartes executions.

4.2 Ecosystem

We will keep aligned the ecosystem with new STAMP tools versions, and we will enhance the support for Gradle build system. This integration effort will take into consideration feedback and expectations from Use Cases requirements and feature requests that will be issued in GitHub, ensuring at the same time that tools will remain compatible with each other in the common integration scenario.

4.3 tutorials, courseware and documentation

We will write a reference documentation about the CI/CD scenario, in order to let STAMP adopters to have a starting point to adapt it to their own needs. We will also setup a new virtual machine containing a Jenkins instance configured to support use cases developed within the CI/CD scenario, which can be used also as course materials for test amplification tutorials. We will then complete the documentation related to the ecosystem, providing STAMP adopters with several tutorial and a STAMP cookbook to describe how to setup variations of the STAMP CI/CD reference scenario and moreover how to use it in different toolboxes.