Title:	WP2 – D2.2 – Initial prototype on configuration test amplification
Date:	November 30, 2017
Writer:	SINTEF, ATOS, XWiki, ActiveEon
Reviewers:	Inria, ActiveEon

Table of Content

1. Executive Summary

This document summarizes the initial achievements in Work Package 2 of the STAMP project, on the configuration testing automation and amplification. The main content of this report has two parts. The first part describes the automatic tools to support people in testing a target software on multiple configurations. The second part describes the experiments on the automatic amplification of testing configurations, and the abstract configuration model we use for the amplification. The document will focus on the design, implementation, and expected usage of the different tools and libraries in these two directions, using the STAMP use cases, or their simplified versions, as examples.

2. Revision History

Date	Version	Author	Comments
11.10.2017	0.01	Hui Song (SINTEF)	Deliverable outline for internal discussion
1.11.2017	0.1	Hui Song (SINTEF)	8, 8.2, 8.3, Appendix
2.11.2017	0.2	Jesús Gorroñogoitia (Atos)	Section 8.1
4.11.2017	0.3	Vincent Massol (XWiki)	Section 7.4
7.11.2017	0.4	Jesús Gorroñogoitia (Atos)	Improvements in Section 8.1
7.11.2017	0.5	Mael Audren (ActiveEon)	Section 7.3
8.11.2017	0.6	Anatoly Vasilevskiy (SINTEF)	Section 7.2
10.11.2017	1.0	Hui Song (SINTEF)	Ready for internal review
21.11.2017	1.1	Hui Song (SINTEF)	Revision after review from ActiveEon
29.11.2017	1.2	Hui Song (SINTEF)	Revision after review by INRIA

3. Objectives

The main objective of Work Package 2 in this period (M6-M12) is to set up the fundamental building blocks for the automatic amplification of testing configurations. In particular, the two major questions we must answer before we start generating new testing configurations are the following.

1. How can developers test their software against multiple configurations in an efficient way
2. How can we interpret the configuration specifications and generate the valid ones based on the amplification results.

In this document, we answer the first question by a configuration testing framework that automatically launches testing processes on multiple configurations, manages the lifecycle of these processes, and aggregates the results. The environment is based on the operation-system-level virtualization of testing configurations, using the Docker container technology. For the second question, we provide an abstract model for testing configurations based on Docker specifications, together with the transformation engines between the model and specifications. We also report the proof-of-concept experiments on the automatic generation of configuration models using constraint solving.

4. Summary of Artifacts

name	section	link	Contact
D22 latest release		https://github.com/STAMP-project/docs-forum/blob/master/docs/d22_config_testing_prototype.pdf	Hui Song (SINTEF)
Configuration testing framework	7.2	The configuration testing tool: https://github.com/SINTEF-9012/config-testing/	Anatoly Vasilevskiy (SINTEF)
Configuration testing as a service	7.3	http://178.170.72.23:8080/ (access credentials on request)	Mael Audren (ActiveEon)

Abstract configuration model	8.1	https://gitlab.ow2.org/stamp/conf-test-ampli/tree/metamodel-atos/	Jesús Gorroñogoitia (Atos)
model-file transformation	8.2	https://gitlab.ow2.org/stamp/conf-test-ampli/tree/metamodel-sintef/dockergen	Hui Song (SINTEF)
Amplification experiments	8.3	https://github.com/SINTEF-9012/ozepy/tree/master/stamp/	Hui Song (SINTEF)
OZ3Py model-based constraint solver	8.3, Appendix	https://github.com/SINTEF-9012/ozepy	Hui Song (SINTEF)

5. Introduction

According to the survey on the state of the practice of configuration testing [1], most of the STAMP partners either always test their software on one particular configuration, or only test a small number of different configurations in special situations, e.g., before major release or after introducing support for new configurations. In the same time, most of the partners reported in the survey that they have experienced bugs that only appear on specific configurations. The following two challenges hinder developers from testing many configurations regularly:

1. It is difficult for developers to design manually many representative candidate configurations. In practice, the users may use the software in many unpredictable ways, and it is hard to cover these different usages by manually designed configurations.
2. It is time- and resource-consuming to prepare the external configurations that are ready for testing. Software configuration includes not only the internal options and parameters written in the configuration file, but also the external environments such as software stacks, resources, external dependencies, architectures, etc. To test the software on a particular external environment, one has to set up the environment properly, including the deployment of platforms, the simulation of resources, the connection of different pieces, etc. This process is expensive in terms of both time and resource.

As a result of the two challenges, it is a common practice that a developer or a quality assurance team member spends some time to set up the entire environment on his/her own working device, and then keep reusing the same environment to test the software.

The main topic of STAMP WP2, the amplification of testing configurations, targets the first challenge: By automatically generating new testing configurations, it is easier to cover more configurations. The ideal case, as we planned in the proposal, is that we generate multiple configuration specifications, and the developers can take these specifications as input to test their system – similar to the idea in WP1, where the generated test cases are used directly as input for the unit-testing framework. However, after surveying the state of the practice, we realized that there is not a widely used "configuration testing framework", the one that takes as input a configuration specification and automatically test the software on this configuration. The reason for this situation is the second challenge as we discussed above: setting up an environment for testing is a complicated task.

An automatic framework to support the testing of multiple configurations is a fundamental building block for the amplification of configuration testing. Considering the situation that most partner do not employ such a framework yet, the first task of Work Package 2 is naturally to build this framework.

In the first part of this document, we report an automatic configuration testing framework. The framework employs the operating-system-level virtualization, in particular the Docker container technology, to simulate the external environment of the target software. During the testing process, the target software is running in a Docker container, which contains the entire software stack required by the testing target, in order to simulate the deployment environment. In addition, when necessary, this main container is connected to several containers that provide the external services required by the software, such as database. The containers are instantiated according to the predefined Docker specifications, in particular, the Dockerfile to specify the deployment environment (Docker images) and the docker-compose.yml file to specify the connection between containers. The framework takes as input a set of Docker specifications, each for a

specific configuration. For each configuration, the framework launches the testing process by building Docker images, instantiating the containers and connecting them together, deploying the target software into the containers, executing the testing scripts, and collecting the report.

We implement the configuration testing framework as a generic testing support tool, integrated into the Jenkins pipeline for continuous delivery. In addition, we provide a Configuration Testing as a Service, by integrating the tool into the online building service hosted by OW2. We have started to use the tool and the online service within WP2 for experimenting the configuration testing amplification. We will also disseminate them to the STAMP partners as well as the development community outside STAMP. In the same time, we foresee that configuration testing can be very different among development teams. For the sake of performance and the compatibility to the existing testing practice, a development team may want to customize or re-implement the environment following the same idea. Therefore, in addition to working together on the generic tool, the XWiki team is also designing and implementing their own configuration testing tool, optimized for, and integrated to, the XWiki testing process. We will also report in this document the architectural design of this tool.

In the second part of this document, we will report our initial experiments on the automatic amplification of testing configurations. Based on the automatic testing framework, and the idea of using OS-level virtualization to simulate testing configurations, we choose the Docker specifications as the amplification target. This guarantees that the amplification results can be directly used as input by our configuration testing framework. We implement configuration amplification as a searching approach: from the provided configuration pieces, such as alternative platforms, parameters and external servicers, the amplifier automatically search for valid ways to put the pieces together into a complete configuration. To ensure that the searched configuration is valid, the approach searches for results within a set of predefined constraints, using a model-based constraint solver.

In this document, we will report the following achievements related to configuration amplification. We first provide an abstract model that captures the key information of testing configurations, and is used as the target model for the searching / constraint solving approach. The models is currently designed according to the Docker concepts, but with the potential extensibility to support other configuration specifications. Together with the abstract model, we also implement the bidirectional transformation between it and the concrete Docker specifications. Finally, we explain how the configuration amplification approach search for valid configuration models, using a simplified example. The search approach employs a model-based constraint solver, which was re-implemented by SINTEF based on their previous tool, specifically for the usage in STAMP. We put as appendix an introduction to this constraint solver.

6.   Acronyms

MDE	Model-Driven Engineering
MDRE	Model Driven Reverse Engineering
ADM	Architecture Driven Modernization
SMT	Satisfiability modulo theory
EMF	Eclipse Modelling Framework

7.   Configuration testing automation

This section presents how to automatically test the target software in multiple configurations, and how to integrate this testing stage into the continuous delivery process. We start the section with a theoretical model of configuration testing automation based on Docker technology and continuous delivery pipelines. After that, we describe three automatic configuration-testing tools, which we design and implement following the theoretical model, but for different application purposes and environments.

7.1. Theoretical model

The theoretical basis of our approach to automating configuration testing is to use the operating-system-level virtualization to simulate the different configurations for testing. OS-level virtualization is a feature in some operating systems, including the latest versions of Linux and Windows, which allows the existence of multiple isolated user-space instances. All the user instances share the same OS kernel, but each of them have an isolated space in terms of both memory and storage. The former allows the isolation between spaces at runtime, while the latter enables that each instance can enjoy its own software stack, except the OS kernel. Container technology, in particular Docker, utilizes the OS-level virtualization feature of both Linux and Windows, to support the build, ship and operation of OS-level virtual machines, i.e., containers. A container is a group of processes at runtime, isolated from other processes with its own memory and storage space. The read-only part of a container can be committed as an image, which can used later on to instantiate new containers. An image can be built either by manually installing software and creating other files, or by a script call Dockerfile. The images can be shipped between different machines as a binary archive or as a script.

We use Docker technology to simulate the following aspects of testing configurations, according our definition of external configurations based on the survey[1].

- **Deployment environment**. A Docker image can include the entire software stack which target software depends on. Different images can be used with alternative platforms or different versions of the same platform. These images can be prepared in advance, so that during the testing process, we do not need to re-install the entire software stack.
- **External dependencies**. Some software requires other software or services, such as databases. These services can run in a separate container, which are linked to the main container with the testing software. These external dependencies can be specified and controlled using a docker-compose file.
- **Resources**. Docker allows the containers to be running under a predefined resource, such as CPU cores, CPU occupation, memory size, etc. We can use this feature to simulate the resource-constraint testing environment.
- **Flexible architecture**. A component-based or microservices architecture system can be running as several Docker containers. We can control the lifecycles of these containers and the links between them to simulate the architectural changes of such systems, such as the failure of nodes, etc.
- **Scalability**. Modern software systems are often horizontally scalable, which use multiple instances of one component to increase the throughput of the entire system. With the help of Docker orchestration tools, such as Swarm, we can simulate the different configurations of such system in order to test the actual performance. For this, we need a physical cluster to run the containers in order to better simulate the network latency between containers.

As an implementation to this theoretical model, we developed a general configuration testing tool, which automatically execute the testing of the target software, on multiple different configurations. These configurations are specified as Docker specifications and simulated by Docker containers. The tool manages the lifecycle of these testing processes, and can be integrated into different software delivery pipelines.
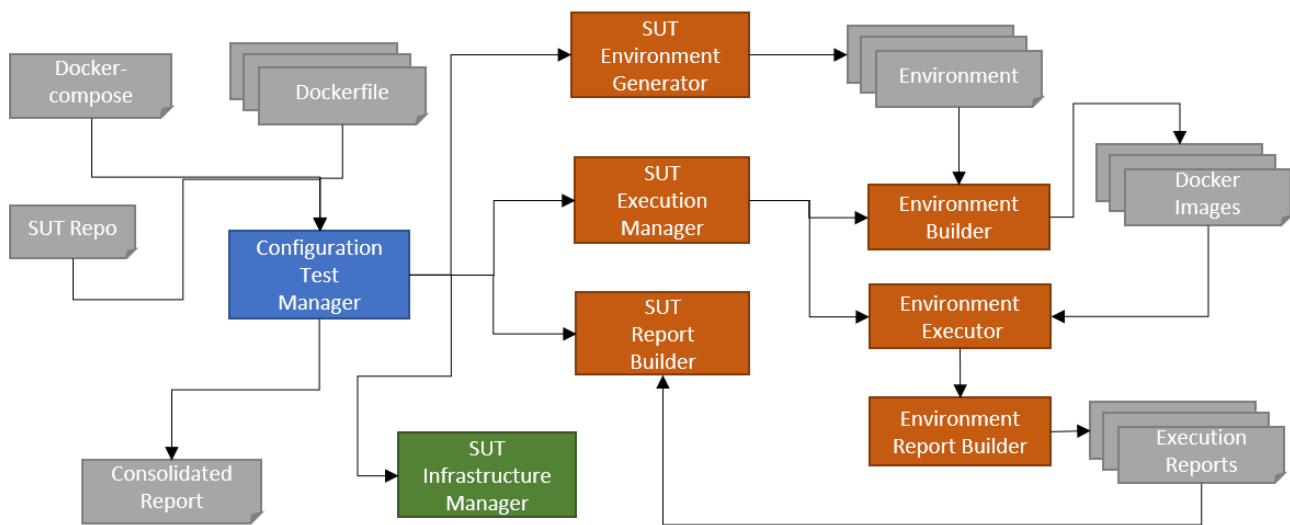
7.2. A generic configuration testing framework

This section describes a conceptual architecture of the Configuration Testing Framework (CTF) and discusses an initial implementation of the proposed architecture. The prototype implementation of the framework is available at the CTF repository[2].

---

[1] Deliverable D2.1 Report on the State of Practices for Configuration Testing, STAMP Deliverable 2.1, 2017,

[2] https://github.com/SINTEF-9012/config-testing

Figure 1 - Conceptual architecture of the configuration testing framework

Figure 1 outlines a conceptual infrastructure of the CTF. The input for the CTF is a set of Dockerfiles, a docker compose file, and a repository of the tool to test (SUT). A docker file represents an environment. The CTF exercises test cases of the SUT against each environment. A docker compose file captures an additional infrastructure which is required for testing. For example, we may require setting up a database and connect a front end of our services to this database before we exercise the SUT. The output of the framework is a consolidated report with the test results in different environments. The CTF conceptually consists of three main modules. In Figure 1 a **plugin framework** is in blue, an **executor plugin** is in orange and an **infrastructure plugin** is in green.

The **plugin framework** implements a plugin infrastructure that allows extending the CTF with different plugins to enable configuration testing for various build systems, e.g. maven, ant, gradle etc. A core element of the plugin framework is the *Configuration Test Manager*. It receives all required input from a user, spawns and controls a test infrastructure and launches a configuration testing for the SUT.

The **infrastructure plugin** allows building pluggable components which can set up a required infrastructure prior a testing of the SUT and kill this infrastructure gracefully once all test cases are executed. The *SUT Infrastructure Manager* is a core component of the infrastructure plugin.

The **executor plugin** executes the configuration testing and assembles all reports for a user. The *SUT Environment Generator* generates and prepares test environments. This step usually requires copying of all required artifacts to build an environment for an execution of the test cases. The *SUT Execution Manager* schedules building of the generated environments and an execution of the test cases in the built environment. The *Environment Builder* takes responsibility of building environments and the *Environment Executor* performs an execution of the test-cases against the built environments. The *Environment Report Builder* assembles results of the test-case execution from a given environment. The *SUT Report Builder* assembles all results in the consolidated report.

Prototype Implementation

To demonstrate different components of the prototype implementation of the CTF, we have enabled configuration testing for some modules of the XWiki platform. This requires some modifications of the XWiki original repository which we have done in a git fork[3]. However, this is not strictly required and has been done to demonstrate the applicability of our approach and the proof-of-concept implementation of the CTF.

---

3 https://github.com/vassik/xwiki-platform
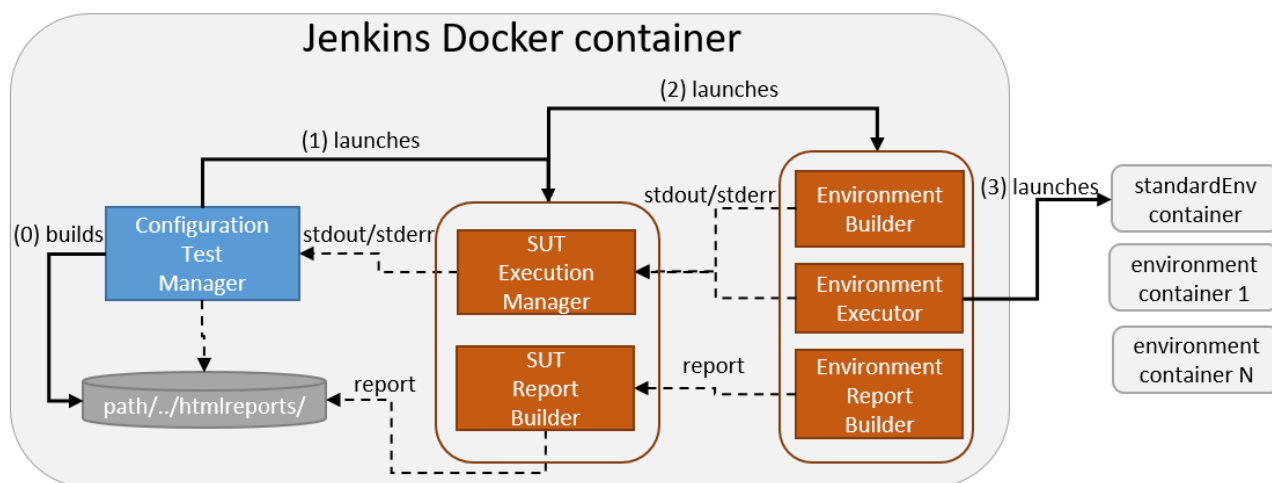
Figure 2 - Prototype implementation of the CTF

Figure 2 shows elements we have implemented in the CTF to enable configuration testing for the xwiki-platform-distribution-flavor-test-misc[4] artifact. In the prototype implementation, the CTF runs inside a Jenkins container and integrates with Jenkins via a Jenkins Pipeline. However, the CTF does not rely on any of the Jenkins functionality and can run independently. A Jenkins file is in the CTF repository and its content is below.

```
node {
    stage('Downloading changes') {
        checkout scm
    }
    stage('Downloading a tool and building') {
        sh "./download_sut.sh"
    }
    stage('Testing') {
        sh "./testframework/test.py"
    }
    stage('Publishing HTML Report') {
        publishHTML (target: [
            allowMissing: false,
            alwaysLinkToLastBuild: false,
            keepAll: true,
            reportDir: 'report/htmlreports',
            reportFiles: 'index.html',
            reportName: "Test Execution Report"
        ])
    }
}
```

Whenever a user triggers a build of the CTF repository, Jenkins downloads from the CTF repository and updates all required configs to execute the configuration testing of the SUT. Consequently, Jenkins executes *./download_sut.sh* which downloads or updates the SUT, i.e. xwiki-platform. Further, Jenkins kicks off *./testframework/test.py* which implements the *Configuration Test Manager* of CTF in Figure 2. Lastly, Jenkins publishes results of the execution of the CTF from a report directory, i.e. *report/htmlreports*, thus a user can navigate to the test results using the Jenkins Web GUI.

---

4 https://github.com/xwiki/xwiki-platform/tree/master/xwiki-platform-distribution/xwiki-platform-distribution-flavor/xwiki-platform-distribution-flavor-test/xwiki-platform-distribution-flavor-test-misc

The CTF repository contains a current implementation of the **plugin framework** as well as *config.ini* which sets up the CTF to test the XWiki platform. The content of *config.ini* is the following:

```
[general]
test_working_folder = work_folder
global_report_dir = ../report/htmlreports
system_under_test = ../xwiki-platform/xwiki-platform-distribution/xwiki-platform-
distribution-flavor/xwiki-platform-distribution-flavor-test/xwiki-platform-
distribution-flavor-test-misc/ ; sut
sut_config_testing_folder = config-testing/ ; relative to %(system_under_test)
```

The config contains a directory where the CTF publishes results. This directory should match to *reportDir* in the Jenkins file to make results available via the Jenkins Web GUI. Also, we specify a directory to the XWiki artifact, i.e. our SUT. The CTF picks up this directory and performs a configuration testing for this module. We also must specify an **executor plugin** to carry out the configuration testing. In our case, the executor plugin is in the repository of the SUT under the *config-testing* directory. The plugin can execute configuration testing for any project that uses maven as a build system. The *Configuration Test Manager* locates and launches the *SUT Execution Manager* in the *config-testing* directory of the *xwiki-platform-distribution-flavor-test-misc* module. In our prototype implementation, we do not generate environments. We add all test environments manually. Therefore, the *Configuration Test Manager* just launches the *SUT Execution Manager*.

Below is the content of the configuration file that the *SUT Execution Manager* consumes for the xwiki use-case.

```
[standardEnv]
category_name = standardEnv
category_test_script = run.py ; %(category_name)s/run.py
```

The configuration file contains a list with all environments to test the SUT. An environment has a name and reference to the *Environment Executor*. In the prototype, each environment is a directory with a name matching to a value of the *category_name* parameter in the configuration file, e.g. *stadardEnv* in the *config-testing* folder. In our implementation, the *Environment Executor* takes a role of the *Environment Builder* and therefore, it also builds the environment, i.e. creates a docker image. The environment bundles scripts which can execute test cases for a given build system. In our use case, we can execute test cases on any environment using the maven build system. The environment also contains configs and files to set up the environment properly. Therefore, we just need to copy a directory with the existing environment, substitute the docker file and finally add a description of the environment in *config.ini* to create a unique environment. Thus, the config file may look as follows:

```
[standardEnv]
category_name = standardEnv
category_test_script = run.py ; %(category_name)s/run.py
[newEnv]
category_name = newEnv
category_test_script = run.py ; %(category_name)s/run.py
```

The *SUT Execution Manager* launches two *Environment Executors*. Each of which kicks off two docker containers. The results of the execution are subsequently returned to the CTF and published by Jenkins.

Installation and usage

Detailed instructions to install the Configuration Testing Framework can be found at the repository[5]. In this subsection, we briefly describe how to install and use the CTF. All required files to build a CTF enabled Jenkins image are in the *jenkins* folder of the CTF repository[6]. We use the following command to build an image:

```
docker build -t jenkins-config_testing:latest .
```

---

5 https://github.com/SINTEF-9012/config-testing/blob/master/testframework/docs/installation.pdf

6 https://github.com/SINTEF-9012/config-testing/tree/master/testframework/jenkins

To launch container from the built image:

```
docker run --restart always --name jenkins-config_testing -d -p 8090:8080 -e
JAVA_OPTS="\"-Dhudson.model.DirectoryBrowserSupport.CSP=script-src * 'unsafe-inline'
'unsafe-eval'; img-src *; style-src * 'unsafe-inline';\"" -e MASTER_SSH_PORT=22 -e
MASTER_SLAVE_USER=jenkins -e MASTER_SLAVE_PWD=jenkins -e DOCKER_GID=993 -v
/var/run/docker.sock:/var/run/docker.sock -v
~/config_testing/jenkins_volumes/jenkins_home:/var/jenkins_home -v
~/config_testing/jenkins_volumes/jenkins_mvn_repo:/var/jenkins_mvn/repo jenkins-
config_testing:latest
```

The CTF performs the configuration testing in separate containers. These containers run alongside with the CTF enabled Jenkins container. Therefore, the Jenkins container needs to have access to the *docker socket* which we share between the Jenkins container and a host machine. In addition, we need to pass a *group id* of the docker group to make sure that the container has access to the shared socket. We share the *jenkins_home* directory with the host machine to persist results and reports.

To set up Jenkins properly, we recommend installing all suggested plugins. In addition, we need the HTML Publisher plugin to make testing results of the CTF available via the Jenkins Web GUI.

Furthermore, we need to create a new GitHub Organization with the links to the repository of the Configuration Testing Framework. Thereafter, Jenkins scans the repository and kicks off the CTF to perform the configuration testing of the module of the XWiki platform.

We can also use the CTF as a standalone tool and initiate a configuration testing from a command line.

```
./deploy.sh -g=<docker_gid> -i=<image_name> -a=<container_name> -
s=<directory_to_report> -t=<tool_to_test>
```

The CTF repository contains the *deploy.sh* script. The script builds an image with the name *<image_name>* form a docker file in the same folder. The script passes *<docker_gid>* to the container *<container_name>*, kicks off the CTF tool to test a tool from *../<tool_to_test>* and makes reports available at *<directory_to_report>/<tool_to_test>*.

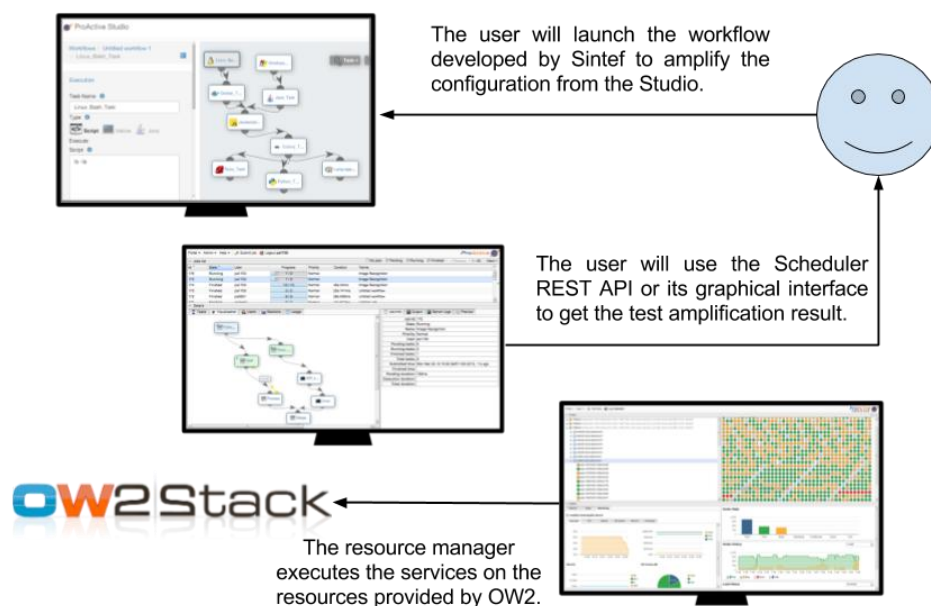## 7.3. Configuration testing as a service

ProActive Workflow and Scheduling developed by Activeeon enables to schedule workflows on infrastructures. The application is composed of several software and configuration testing requires the use of three of them.

- The studio interface allows to build workflows.
- The scheduler orchestrates and automates multi-application jobs.
- The resource manager interface manages and automates the resource provisioning from infrastructures.

The application run on a virtual machine on OW2Stack which is the OpenStack platform provided by OW2 and use this platform as a resource to schedule jobs.

Sintef uses the studio to move from Jenkins pipeline to Activeeon workflows in order to execute configuration testing as a service. The workflow is composed of four scripts executed one by one:

1. Configuration testing tool bootstrap
2. Download SUT which is the configuration testing tool
3. Execute the configuration testing
4. Write the reports

The user will launch the workflow developed by Sintef to amplify the configuration from the Studio.

The user will use the Scheduler REST API or its graphical interface to get the test amplification result.

The resource manager executes the services on the resources provided by OW2.

7.4. The XWiki configuration testing environment

The XWiki project currently has automated functional tests that use Selenium and Jenkins. However they exercise only a single configuration: HSQLDB, Jetty and FireFox (and all on a single, fixed version).

Work was done in STAMP to provide testing on various configurations. A first step was achieved by developing several Docker images for XWiki. Right now the following different environments are supported:

- LTS version of XWiki, on latest Tomcat 8, MySQL (any version >= 5.7)
- LTS version of XWiki, on latest Tomcat 8, PostgreSQL (any version >= 9.5)
- Latest stable version of XWiki, on latest Tomcat 8, MySQL (any version >= 5.7)
- Latest stable version of XWiki, on latest Tomcat 8, PostgreSQL (any version >= 9.5)

These images and the build for them can be found at https://github.com/STAMP-project/docker-xwiki.

We were also able to propose and have the XWiki Docker images accepted as official images on DockerHub: https://hub.docker.com/_/xwiki/ (and as of today they count already 500K+ pulls).

However this is only a first step since these images are currently not exercised by the XWiki functional tests. Thus we're proposing below an architecture that should allow XWiki to be tested on various configurations:

- Various supported databases and versions
- Various Servlet containers and versions
- Various Browsers and versions
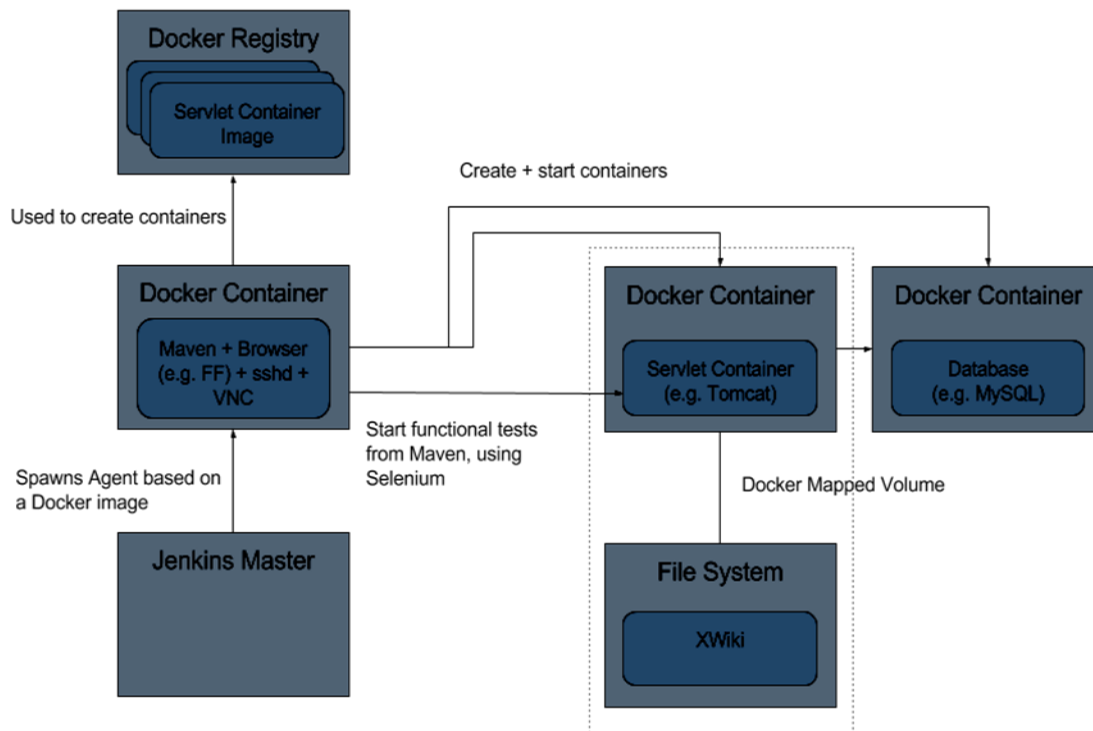
Here's what it would mean for implementing it as a Jenkins Pipeline (note that at this stage this is pseudo code and should not be understood literally):

```
pipeline {
  agent {
    docker {
      image 'xwiki-maven-firefox'
      args '-v $HOME/.m2:/root/.m2'
    }
  }
  stages {
    stage('Test') {
      steps {
        docker.image('mysql:5').withRun('-e "MYSQL_ROOT_PASSWORD=my-secret-pw"') { c ->
          docker.image('tomcat:8').withRun('-v
$XWIKIDIR:/usr/local/tomcat/webapps/xwiki').inside("--link ${c.id}:db") {
            [...]
            wrap([$class: 'Xvnc']) {
              withMaven(maven: mavenTool, mavenOpts: mavenOpts) {
                [...]
                sh "mvn ..."
              }
            }
          }
        }
      }
    }
  }
}
```

Some explanations:

- We would setup a custom Docker Registry so that we can prepare images that would be used by the Jenkins pipeline to create containers
- Those images could themselves be refreshed regularly based on another pipeline that would use the docker.build() construct
- We would use a Jenkins Agent dynamically provisioned from an image that would contain: sshd and a Jenkins user (so that Jenkins Master can communicate with it), Maven, VNC Server and a browser (FireFox for ex). We would have several such images, one per browser we want to test with.
- Note that since we want to support only the latest browsers versions for FF/Chrome/Safari we could use apt to update (and commit) the browser version in the container prior to starting it, from the pipeline script.
- Then the pipeline would spawn two containers: one for the DB and one for the Servlet container. Importantly for the Servlet container, I think we should mount a volume that points to a local directory on the agent, which would contain the XWiki exploded WAR (done as a pre-step by the Maven build). This would save time and not have to recreate a new image every time there's a commit on the XWiki codebase!
- The build that contains the tests will be started by the Agent (and we would mount the the Maven local repository as a volume in order to sped up build times across runs).
- Right now the XWiki build already knows how to run the functional tests by fetching/exploding the XWiki WAR in the target directory and then starting XWiki directly from the tests, so all we would need to do is to make sure we map this directory in the container containing the Servlet container (e.g. in Tomcat it would be mapped to [TOMCATHOME]/webapps/xwiki).

Next step is to implement it and start testing various configurations through a Jenkins pipeline setup.

8. Configuration model and amplification

This section reports a simple model abstracting the essential information in the software configuration, as well as our initial attempts to amplify testing configurations based on this model. The figure below shows an overall structure of this work.
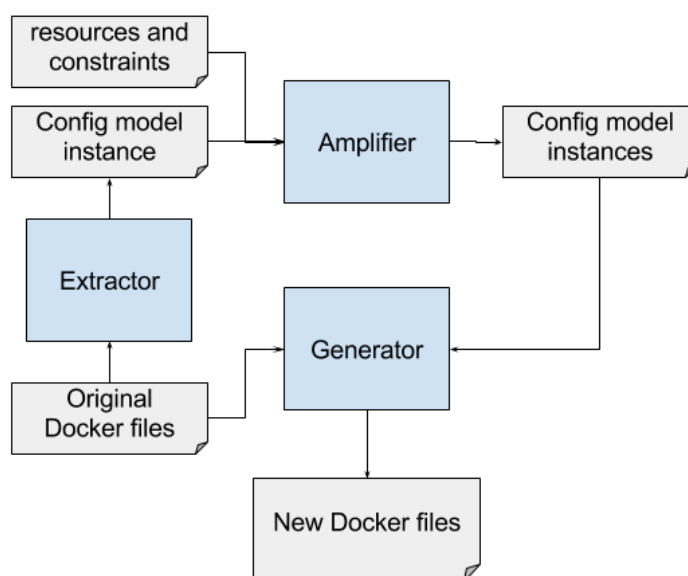
Both the input and the output of this approach are docker files, in particular, Dockerfile specifying images and docker-compose.yml file specifying the docker containers and the connections between them. Each set of docker files (one docker-compose.yml together with 0 to multiple Dockerfiles) describes a potential configuration to test the target system.

The approach takes as input one or a few configurations, each of which is specified by a set of docker files. The extractor transforms the input seed into an abstract configuration model, as the input to the amplifier. In

addition to the seed model, developers need to provide the alternative pieces, such as other images, as well as the relations between these pieces. So far, the resources and constraints are specified on the abstract model. In the next step, we will investigate how to annotate this information directly on the docker files. The extracted configuration model will be used as input by the **amplifier**, which produces a number of new configuration models. Each of these models represent a unique configuration. The generator transform these models back into docker files. The extractor and the generator together form the **bidirectional transformation engine**.

In the rest of this section, we briefly describe the initial versions of the configuration model, the bidirectional transformation engine and the amplifier.

8.1. An Initial configuration model for testing amplification

This section describes an initial attempt to abstract the modeling of test configurations, particularly paying attention to the modeling of the mutability points of such configurations. The modeling initiative describe hereafter adopts the following initial set of assumptions and justifications:

- The main adopted domain specific technologies for describing test configurations are Docker (I.e. Dockerfile) and Docker Compose (I.e. docker_compose.yml). As a matter of fact, these test configuration technologies are widely adopted by STAMP UC providers, as confirmed by the results of a previous survey, reported in D2.1[7]
- In order to design and implement a test configuration amplification framework that supports any arbitrary test configuration technology, the modeling approach should enable raising the abstraction model up from the ground technologies, so the amplification is driven in a technology independent manner.

Satisfying second assumption is possible thanks to the adoption of modeling abstraction techniques, in particular the OMG ADM horseshoe model[8].

---
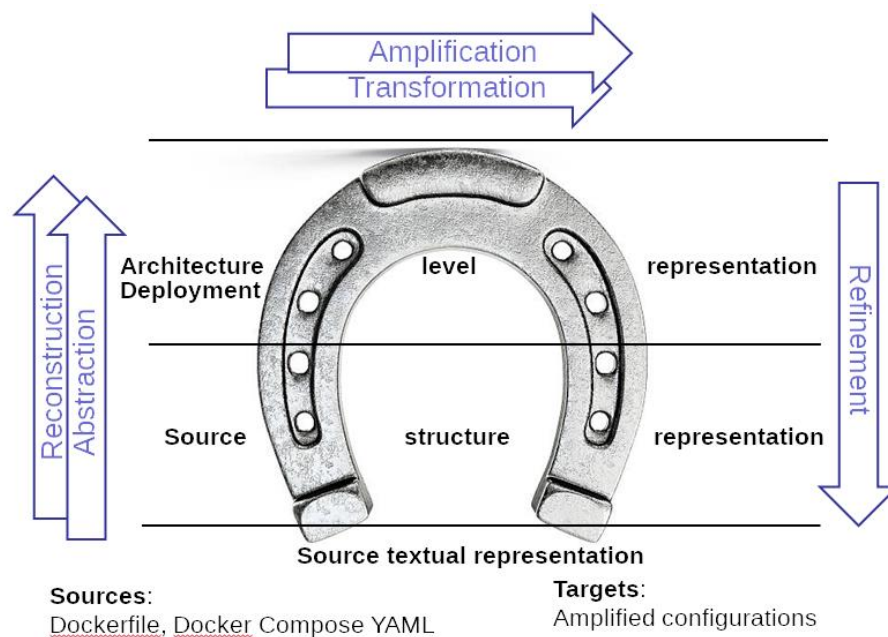
[7] Deliverable D2.1 Report on the State of Practices for Configuration Testing, STAMP Deliverable 2.1, 2017,

[8] http://adm.omg.org/ADMRoadmapv6_VK.pdf

Figure 8.1.1: OMG ADM horseshoe model

The amplification of existing test configurations can be considered as a specialization of the architecture-driven modernization (ADM) process; therefore, it can be addressed by applying the ADM horseshoe model. Pre-existing test configurations describe predefined architectures for the current software under test. By test configuration amplification, we mean the generation of new test configuration architectures, through steps that can be mapped to those in the ADM modernization process. These configurations can be mapped to the legacy source representations mentioned in the Figure 8.1.1. Hence, the horseshoe model defines a configuration test amplification cycle that:

- Starts from preexisting test configurations (e.g. Dockerfiles) that are source text representations of the legacy configurations;
- Raises the abstraction representation of these configurations up to the architecture representation;
- Operates at this abstract architecture level representation to compute additional (e.g. compatible and/or optimized) test configuration models, by applying model to model transformations, and;
- Synthesizes new amplified test configurations out of these transformed architecture models, through a refinement process that leverages on model to text transformations.

The remaining of this section describes the current modeling approach conceived to obtain a high-abstraction architecture representation of the preexisting test configurations, that is, the left half side of the horseshoe. The approach adopts Model Drive Reverse Engineering (MDRE) technologies and consists on:

- Parsing an input test configuration descriptor (e.g. Dockerfile or docker_compose.yml files) to to create an intermediate in-memory representation of certain relevant elements of these configurations;
- Generating an abstract architecture configuration model instance, which is compliant with an architecture configuration metamodel, out of the parsed model.

Configuration Metamodel

The configuration metamodel is designed to capture the platform independent test configuration properties, and in particular the variability points of the configuration. Variability points, named mutability points hereafter, declare the elements in the configuration that are amenable to be modified during the test configuration amplification process.

Figure X represent the Configuration metamodel. This metamodel refers to another, namely the Feature Taxonomy metamodel, depicted in Figure 8.1.2, that provides a classification of features that qualify the domain-specific semantics of some concepts included in the configuration metamodel.
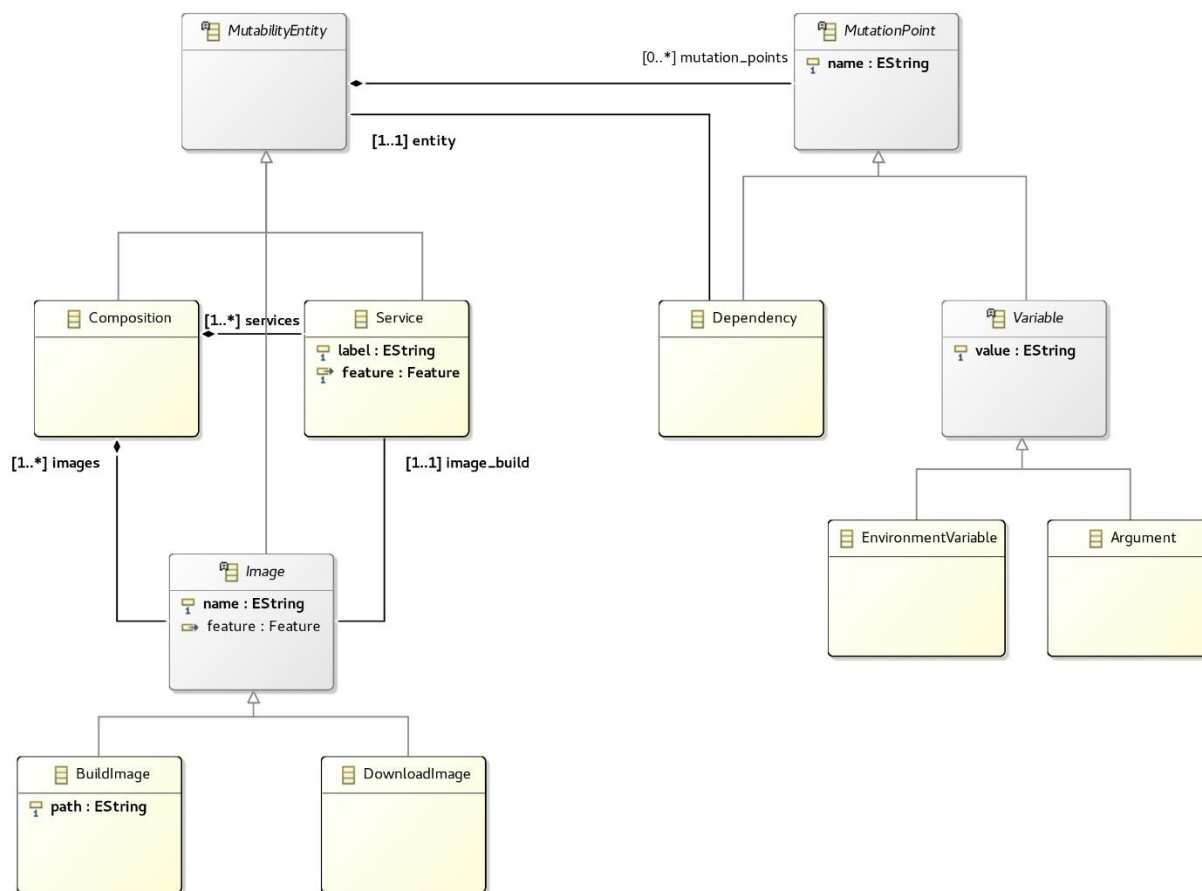
Figure 8.1.2: Configuration Metamodel

A configuration is possible for individual services or compositions of them. Focusing on a wider usage of this metamodel, let's start from a composition, as an aggregation of services. Grounding to the technology we based our modeling abstraction endeavor on, a composition is equivalent to a Docker Compose aggregation of containers. Each service in the composition is described by an image, which can be either a downloadable image or an image that can be built from an existing image descriptor (e.g. our technological grounding counterpart is a Dockerfile), located at the given path, relative to our composition descriptor. In order to enable the existence of different services built from reusable images, a composition may also contain referenced images.

So far, described metamodel captures the essential configuration metamodel elements. Next step consists on introducing the capability to model variability in the configuration. Variability is introduced by identifying mutable entities, subclasses of an abstract MutabilityEntity, which qualifies these mutable elements of the configuration. Being generic enough, both the composition itself, aggregating services and the images they are built from are mutable entities, as they can be either parameterized or replaced by other similar entities.

Each of these mutable entities is characterized by 1 or more mutation points, labeled by name. Two mutable points have been identified:

- A variable: represents a key, value tuple. Two types of variables are declared:
  - An environment variable, whose valued is passed within the execution environment, so having effect at runtime, and;

o   An argument variable, whose valued is passed within the configuration (I.e. default value), or during the configuration enactment, through the configurator API;

- A dependency: constitutes a causal connection between one mutable entity an another, so the operation of the former depends on the availability of the latter.

Mutable entities could be:

- The composition itself, which could be parameterized with arguments that modify either its structure or its behavior (for instance, its non-functional properties);
- A service, which could be similarly parameterized, through environment variables, for example, or by parameterizing its available resources. Besides, a service may depend on a number of other services, through dependency relationships.
- An image, which could be similarly parameterized, by issuing arguments that are passed to the image descriptor during its building process.

Feature Taxonomy metamodel

Some elements in the metamodel are functionally qualified by a taxonomy of features whose metamodel is declared aside and imported by the configuration metamodel. A feature qualifies a functional (but even a non-functional) characteristic of some configuration elements, namely:

- A Service: declares a concrete functionality, which can be associated by a pointer to a concrete feature (or a superclass of similar features) in the taxonomy.
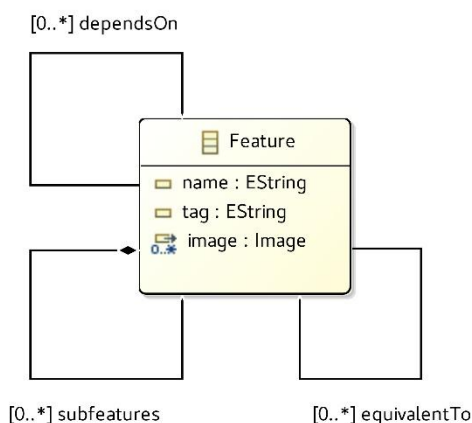- An image: provides a concrete implementation of a feature in the taxomony.

Figure 8.1.3: Feature Taxonomy metamodel

Figure 8.1.3 represents the feature taxonomy metamodel. This metamodel enables the modeling of a tree-shape taxonomy of features. From the root feature (e.g. the taxonomy entry point) other taxonomy branches can be declared using the subFeature reference. Two features can be declared equivalent using the equivalentTo reference (i.e. depending on feature semantics this equivalence can be functional or non-functional). Similarly, a feature can be declared depending on others, using the dependsOn references.

Note that two child features located under the same parent feature are not assumed to be equivalent (I.e. they are specializations of the feature parent, so only equivalents on what concerns the parent characterization), so this explicit relation is required when this characteristic has to be made explicit. For instance, MySQL, MariaDB and PostgreSQL are subfeatures of a RDBMS feature parent, but only MySQL

and MariaDB features would be equivalent in terms of compatibility, although they all three are SQL-based database engines.

A feature is characterized by a name and a tag, which specifies it deeper (e.g. release, version, flavor, etc.). Eventually, a feature could reference known images that implements it.

Configuration and Feature Taxonomy Editors

Both configuration and feature taxonomy metamodels have been authored using Eclipse Modeling Framework (EMF) under Eclipse Oxygen (v4.7). They are included in the EMF project *eu.stamp.configuration.metamodel* located in the STAMP GitLab repository *conf-test-ampli*, under the *metamodel-atos* branch, in the model folder:

https://gitlab.ow2.org/stamp/conf-test-ampli/tree/metamodel-atos/eu.stamp.configuration.metamodel

The configuration metamodel is defined in the *deployment_configuration.ecore* file. The feature taxonomy metamodel in the *feature_taxonomy.ecore*.

EMF technology generates editors for authoring model instances compliant with a given metamodel. Using this technology, we have generated an Editor for modeling configuration and feature taxonomies models. EMF generates a number of source projects from the given metamodel that, through a compilation and building process, generate a number of plugins that can be deployed in any compatible Eclipse installation, enabling its users to author their configuration and feature taxonomy models. In the STAMP GitLab repository:

https://gitlab.ow2.org/stamp/conf-test-ampli/tree/metamodel-atos

Those projects are located:

- eu.stamp.configuration.metamodel: contains above described metamodels and the generated Java sources for the metamodel classes and relationships.
- eu.stamp.configuration.metamodel.edit and eu.stamp.configuration.metamodel.editor: contain the metamodel editor source code.

This editor can be deployed into an existing Eclipse installation through the Eclipse export facility. Select above projects in the Navigator Explorer, right-click and select *File/Export* in the contextual menu. Then, select *Plug-in Development/Deployable plug-in and fragments* in the wizard. On next wizard page, select the dropins folder of the Eclipse destination in the Destination tab.

Start the target Eclipse installation.

To create a configuration model, in Eclipse top menu, select *File/New* and in the wizards list, *Example EMF Model Creation Wizards/Deployment_configuration Model*. To create a feature taxonomy model proceed as before and select the *Feature_taxonomy Model* in the wizard.

Examples of Configuration and Feature Taxonomy models

Figure X shows a snippet of the configuration model for the Atos SUPERSEDE UC, which has been authored manually, using the generated EMF editor. As shown in the editor, the configuration represents a composition of different SUPERSEDE services, each of them instantiated from a downloadable image (as shown in the properties tab for selected postgres service). Dependencies among services are also captured in the model. Services and images in this model are annotated with features classified in a taxonomy modeled in Figure X.
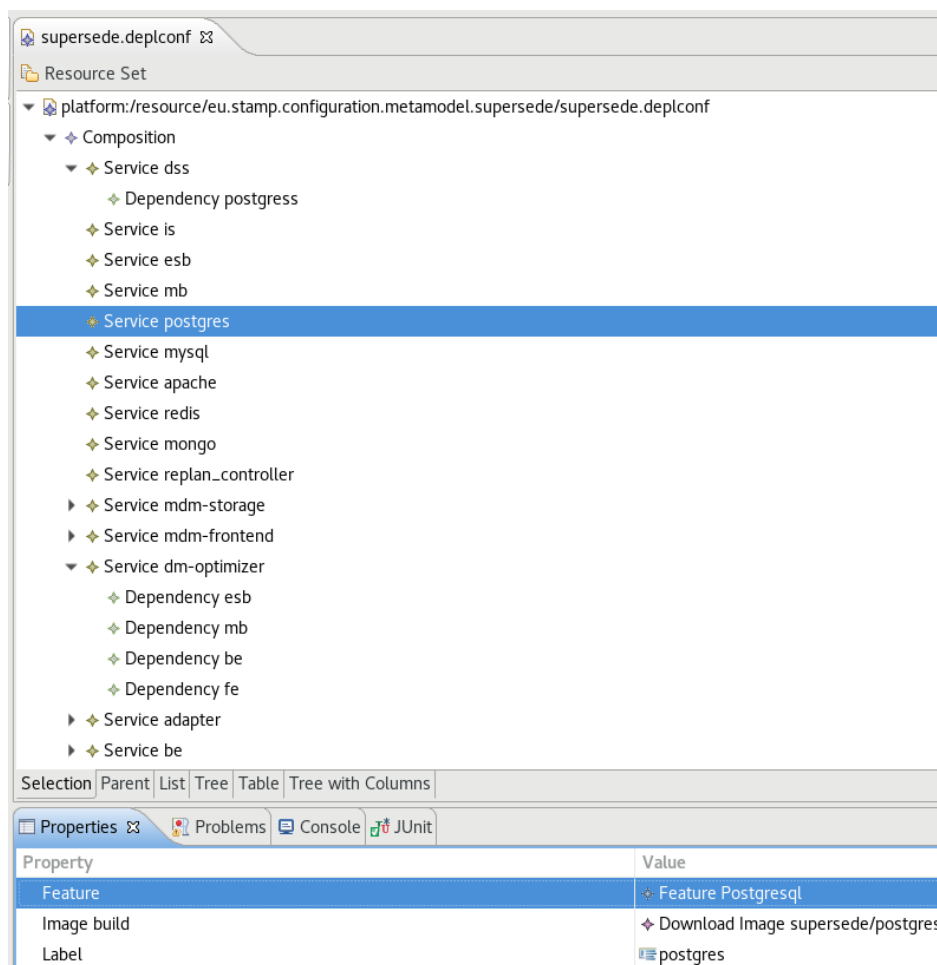
Figure 8.1.4: configuration model instance for the Atos SUPERSEDE UC Docker composition descriptor.

Figure 8.1.4 shows a snippet of the taxonomy of features for the Atos SUPERSEDE UC domain. They represent the features and their classification that are relevant in this domain. As an example of an equivalence declaration is shown this equivalence between MariaDB and MySql in the properties tab.
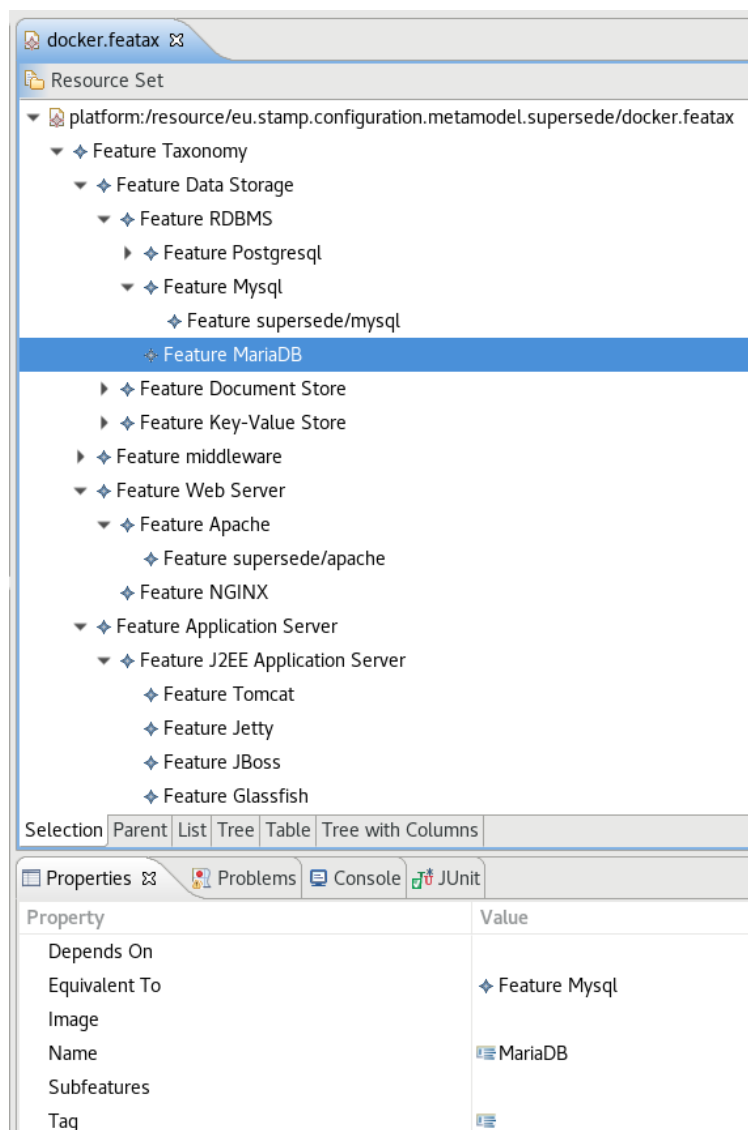
Figure 8.1.5: Feature Taxonomy model instance for the Atos SUPERSEDE UC domain.

Configuration abstraction from baseline deployment descriptors

As afore mentioned when describing the ADM horseshoe cycle, preexisting test deployment configurations have to be abstracted up to the level of model instances compliant with the configuration metamodel. This abstraction process can be achieved by applying MDRE techniques, depicted in the Figure 8.1.6:
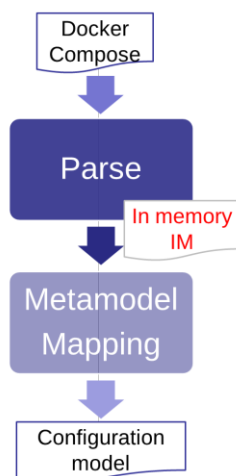
Figure 8.1.6: MDRE process for configuration abstraction from baseline deployment descriptors

In the first step, a parser takes as input the Docker Compose descriptor and populates an in-memory Intermediate Model (IM) that captures relevant information from the descriptor. In the second step, a metamodel mapping process generates a model instance compliant with the configuration metamodel, by mapping elements of the IM with the elements of the metamodel.

A **Docker Compose Parser** has been implemented in Java using Regular Expressions. Source Code is available at: https://gitlab.ow2.org/stamp/conf-test-ampli/tree/metamodel-atos/eu.stamp.configuration.metamodel.parser

Current prototype loads the input Docker Compose YAML descriptor and uses regular expressions to capture from it information about declared services, their dependencies and associated images, creating a POJO-based in-memory IM, which consists on a collection of services, whose model template is shown in Figure 8.1.7. The parser captures essential information from the compose descriptor, namely for each service: a) its list of dependencies on other declared services, b) the image (downloadable or built from descriptor) the service container is launched from, and c) the mutation points declared in the case of an image build descriptor.
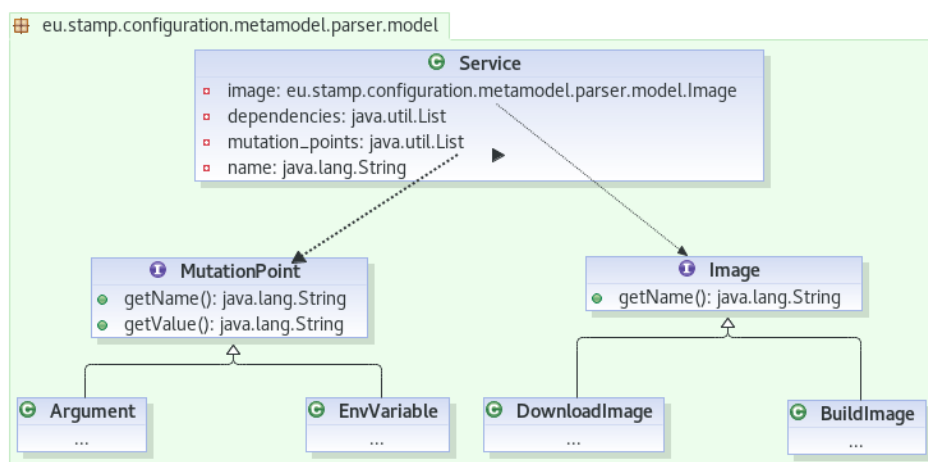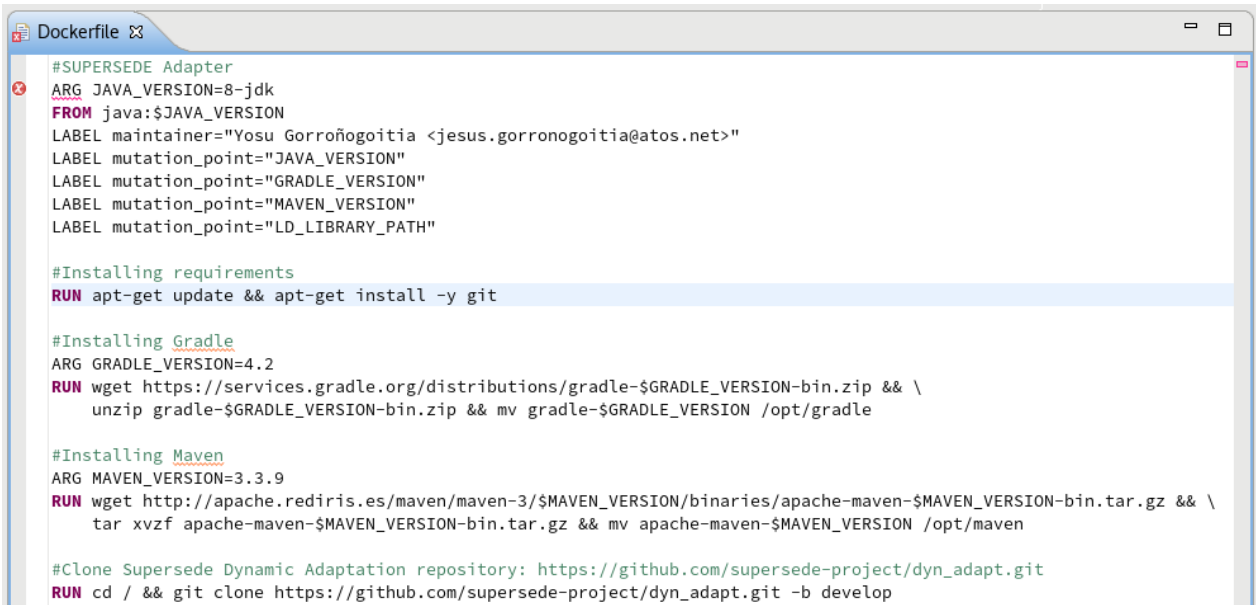

Figure 8.1.7: Parser IM

When the parser matches a service in the compose descriptor whose image is built from a Dockerfile, this descriptor is also parsed, looking for mutable entities declared within, which are attached to the IM service placeholder (I.e. mutation_points collection).

```
Dockerfile

#SUPERSEDE Adapter
ARG JAVA_VERSION=8-jdk
FROM java:$JAVA_VERSION
LABEL maintainer="Yosu Gorroñogoitia <jesus.gorronogoitia@atos.net>"
LABEL mutation_point="JAVA_VERSION"
LABEL mutation_point="GRADLE_VERSION"
LABEL mutation_point="MAVEN_VERSION"
LABEL mutation_point="LD_LIBRARY_PATH"

#Installing requirements
RUN apt-get update && apt-get install -y git

#Installing Gradle
ARG GRADLE_VERSION=4.2
RUN wget https://services.gradle.org/distributions/gradle-$GRADLE_VERSION-bin.zip && \
    unzip gradle-$GRADLE_VERSION-bin.zip && mv gradle-$GRADLE_VERSION /opt/gradle

#Installing Maven
ARG MAVEN_VERSION=3.3.9
RUN wget http://apache.rediris.es/maven/maven-3/$MAVEN_VERSION/binaries/apache-maven-$MAVEN_VERSION-bin.tar.gz && \
    tar xvzf apache-maven-$MAVEN_VERSION-bin.tar.gz && mv apache-maven-$MAVEN_VERSION /opt/maven

#Clone Supersede Dynamic Adaptation repository: https://github.com/supersede-project/dyn_adapt.git
RUN cd / && git clone https://github.com/supersede-project/dyn_adapt.git -b develop
```

Figure 8.1.8: Docker file example with mutation_point anotations

To declare mutable entities, namely, arguments (ARG) and environment variables (ENV), users can use the metadata LABEL *mutation_point* declaration. Figure X shows an example of Dockerfile annotated with metadata for declaring four mutation_points: J*AVA_VERSION, GRADLE_VERSION, MAVEN_VERSION* and *LD_LIBRARY_PATH*, which refers to ARGs or ENV parameters declared and used elsewhere in the descriptor.

This parser will be use, as a library API, during the next metamodel mapping step to create a configuration model instance. In order to create this library, execute the following command in a CMI positioned at the parser project root directory:

```
mvn clean package
```

As a result, the library file *configuration.metamodel.parser-0.0.1-SNAPSHOT.jar* is placed in the folder target.

This parser is tested in the JUnit test *eu.stamp.configuration.metamodel.parser.ParserTest.java. It includes 2 tests: testParseComposition* and *testParseDockerfile,* which take as an input Docker Compose and Dockerfile descriptors, respectively, from the Atos SUPERSEDE UC project *eu.supersede.platform.docker*, which has to be present in the same Eclipse workspace. This project is available at STAMP GitLab: https://gitlab.ow2.org/stamp/atos-uc-supersede/tree/master/docker

A **Configuration Generator** has been implemented in Java as an Eclipse Plugin using EMF. Source Code is available at: https://gitlab.ow2.org/stamp/conf-test-ampli/tree/metamodel-atos/eu.stamp.configuration.metamodel.generator

This generator applies mappings between the IM and the configuration metamodel to create a compliant model instance, out of the input Docker compose generator. This generator performs the complete

configuration abstraction process depicted in Figure X, since before generating the configuration model instance, it invokes the parser to obtain the IM.

It can be tested using the JUnit test *eu.stamp.configuration.metamodel.generator.test.ConfigurationGeneratorTest.java*. The generated configuration model instance is placed within the same location as the input Docker Compose descriptor. Figure 8.1.9 shows a snippet of the generated model.
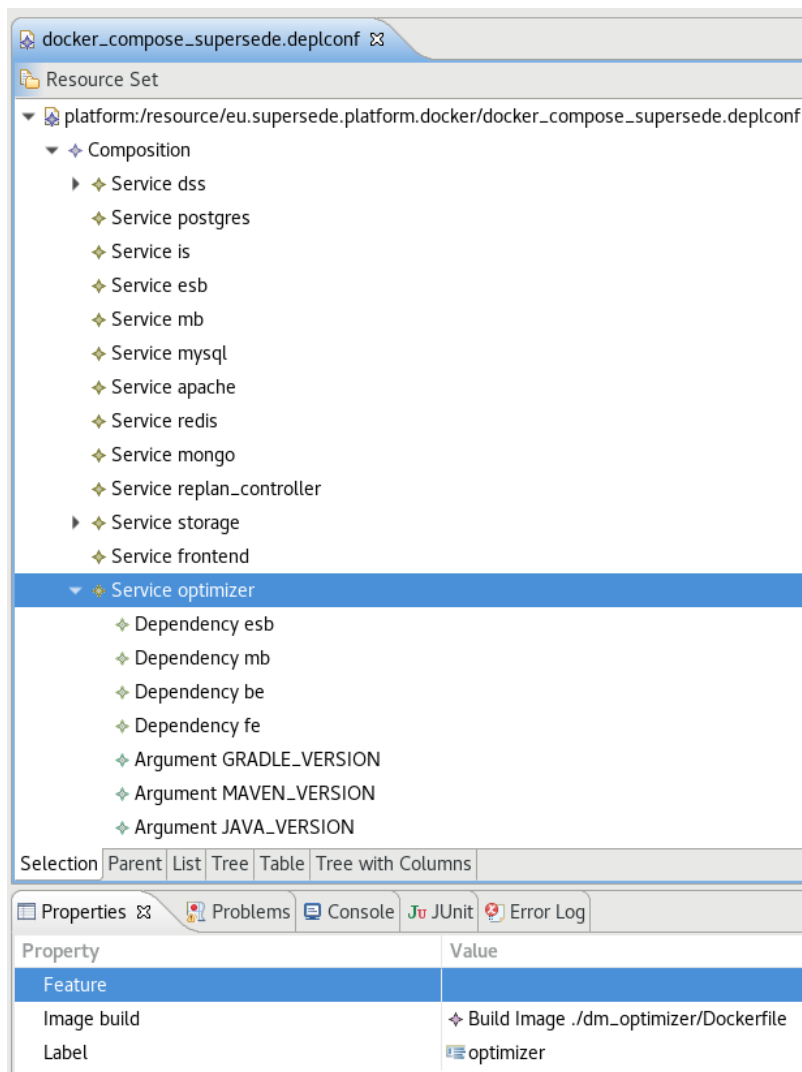

Figure 8.1.9: Generated Configuration model instance

Configuration Generator deployment and usage

The Configuration Generator plugin can be installed into an Eclipse instance following the following procedure (tested in Eclipse Oxygen v4.7.1a):

The following requirements must be installed in the computer:

- Eclipse Oxygen v4.7.1a
- Apache Maven 3.3.9 (or above)
- Git

Installation procedure:

1. Clone STAMP GitLab repository (it creates a local Git repository):

git clone https://gitlab.ow2.org/stamp/conf-test-ampli/tree/metamodel-atos -b metamodel-atos

2. Import the projects located in above local Git repository into the Eclipse workbench, as described above in the procedure to install the Configuration metamodel editor
3. Export the following projects into the target Eclipse dropins folder, as explained above in the afore mentioned procedure (see Figure X snapshot of Eclipse Deployable wizard):
   a. eu.stamp.configuration.metamodel
   b. eu.stamp.configuration.metamodel.edit
   c. eu.stamp.configuration.metamodel.editor
   d. eu.stamp.configuration.metamodel.generator
4. Restart the target Eclipse.

Note: in next releases, an Eclipse Update Site will be provided, what largely facilitates the installation of plugins without requiring to install them from sources.
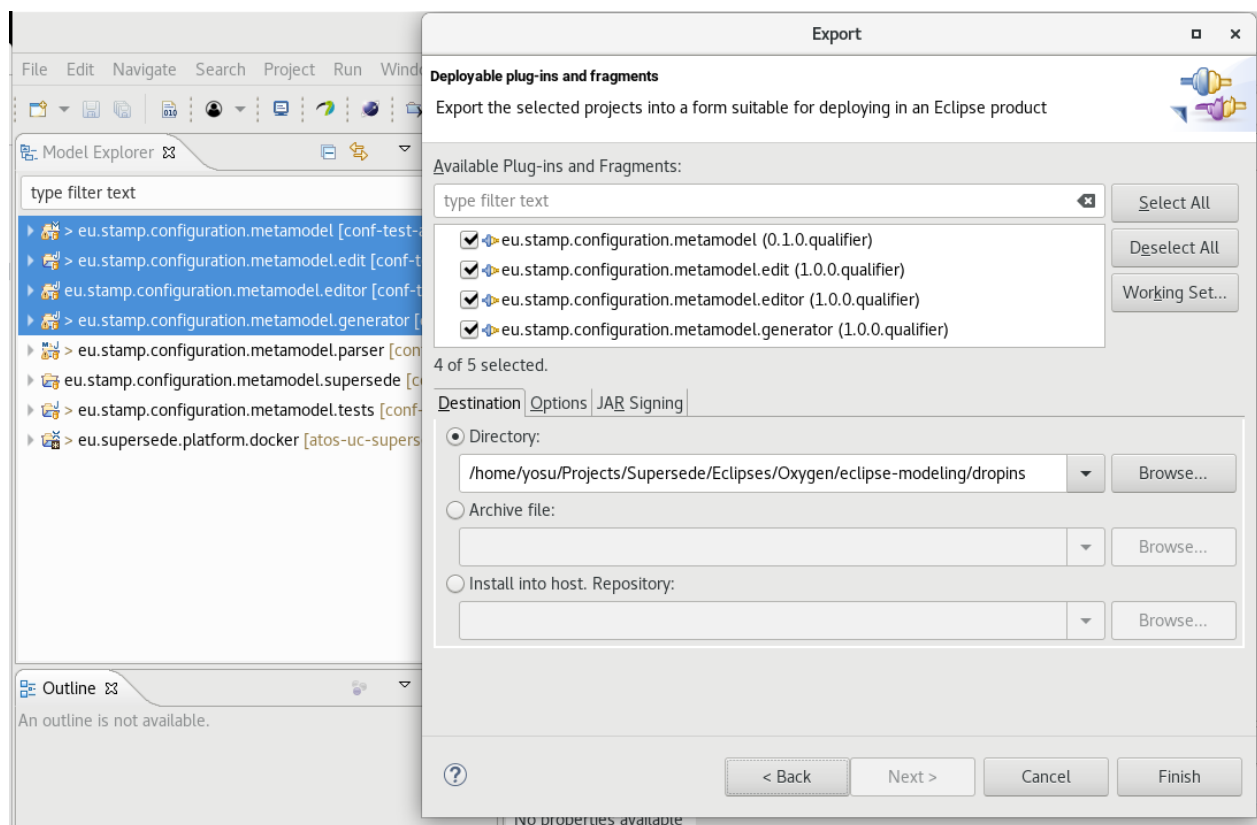

Figure 8.1.10: Snapshot of Eclipse Plugin deployment wizard

Usage procedure:

• Select a Docker File YAML descriptor in the Project Explorer. Right click and choose STAMP/Generate Configuration Model menu entry (see Figure X).
• The generated configuration deployment model will appear in the Project Explorer and also opened into the default EMF editor.
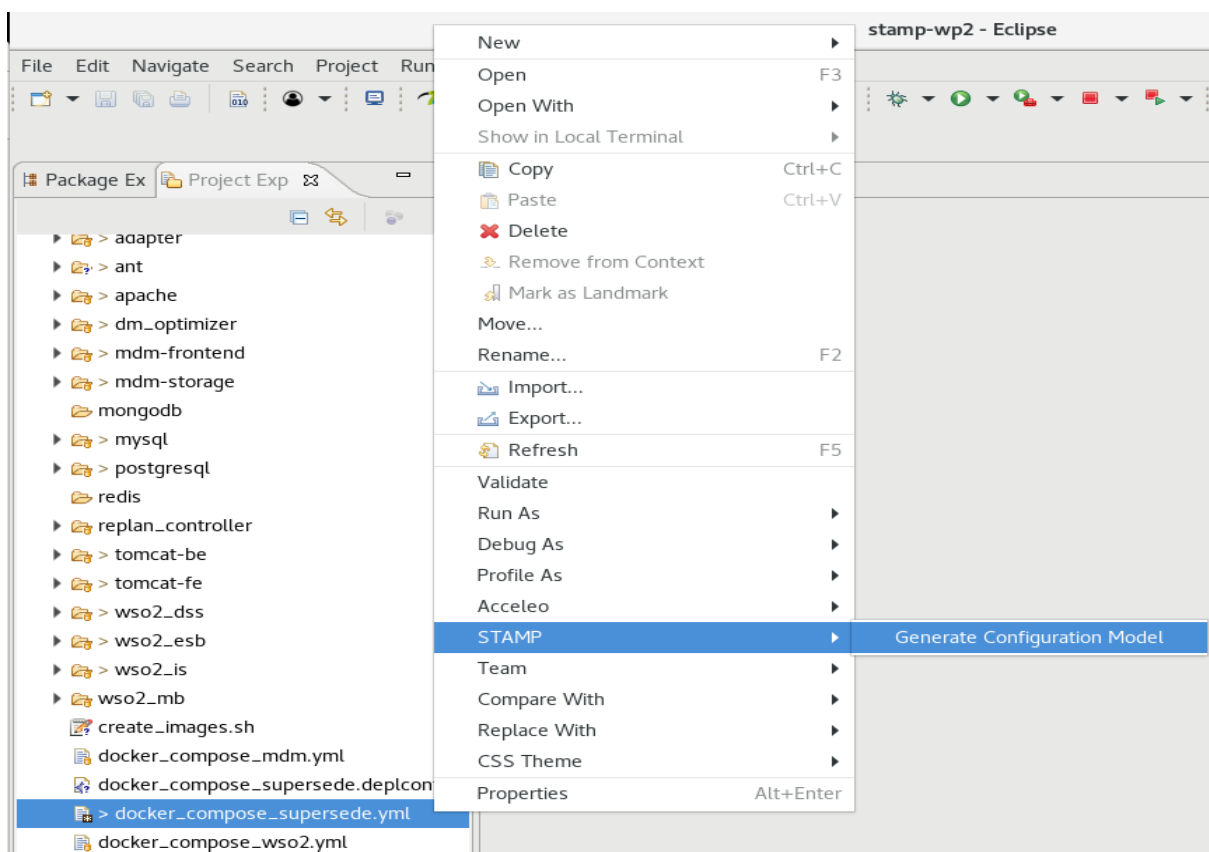
Figure X: STAMP context-menu for Configuration generation

8.2. Transformation between the configuration model and Docker specifications

We developed two bidirectional transformation engines between the docker specifications and the abstract model, one for generating images and another for generating contain orchestrations. Both can be found in the repository: https://gitlab.ow2.org/stamp/conf-test-ampli/tree/metamodel-sintef/dockergen

Dockerfile generator

This generator takes the following two inputs:

- A repository of existing Dockerfiles, each in a separate folder together with all the resources needed by it.
- An amplified model containing a number of stacks. Each stack indicate how to build a new image from the provided ones.

The output is a new repository of generated Dockerfiles, each of which in a folder with all the required resources. A build.sh script is also generated, so that a simple command bash ./build.sh could build all the images.

A sample usage is as below:

```
python src/dockerfilegen.py -i samples/images/java-python/result1.yml
```

By default, the working directory is where that the input file locates. In the directory, the "repo" folder contains the input Dockerfiles while the "build" folder contains the generation results.

The Dockerfile generator employs a typical model-to-text transformation technique: The input model is a number of image chains, each of which describes how an expected image is constructed, step by step, from a downloadable image. The generator iterates all "construction" step, i.e., building one image from another one. It reads the original Dockerfile corresponding to the building rule, check the text content line by line, and change the relevant lines. In the initial version, the only line relevant is the one for "FROM …", The generator change the name of the base image according to the building step.

Docker compose generator

This generator takes one input, i.e., a seed docker-compose file, and generate a number of new files.

```
python src/composegen.py -i samples/compose/atos/docker-compo
```

The generator first extract an abstract model from the input compose. We call this model *m0*

The model only contains the information that is interesting to the amplifier. So far it includes only the services, the service name, the image name and the depends_on between services. From *m0*, the amplifier will produce a number of new models, namely *m1*, *m2*, ..., *mn*. In the example, we have three "generated models", as can be seen from Line 33 of src/composegen.py. After that we merge each generated model back into the docker-compose file, based on "three-way comparison" and generate *n* new docker files.

The generator understands and produces the docker-compose.yml file using a YAML library named PyYAML. The library loads the text file in YAML syntax into a dictionary in Python, and after the generator produce a number of mutated dictionaries, it dumps each of them into a new text file.

One of the main challenge behind of the bidirectional transformation is that the abstract model does not contain all the information in the original docker specifications, but only the information that is interesting to the amplifier. However, when transforming the abstract model back into the docker specifications, the lost information has to be recovered in order to yield valid specifications. This is a typical challenge for bidirectional transformation. We introduce a so called "three-way comparison to address this issue. The basic idea is that we use the original docker-compose file as the basis, and then by comparing the original abstract model with the generated one, we calculate what is changed by the amplifier and do the corresponding changes on the docker-compose file.

Specifically, the current transformation contains the following steps:

1. We find all the services that are in the original abstract model but not the generated one. For each of these services, we find the service with the same name in the docker-compose file, and remove it.
2. For any service in both the original abstract model and the generated one, but with attributes changed, either image or dependencies, we change the corresponding service in the docker-compose file, on the same attributes.
3. For any service in the generated abstract model but not the original one, we generate a service and insert it into the docker-compose file. However, this service only contains the basic information, i.e., the image and dependencies.
4. For any service from the docker-compose file which exists in neither the original nor the generated abstract model, we keep it as is.

8.3. Configuration amplification based on constraint solving

This section reports the initial experiments on the automatic generation of new testing configurations from the ones provided by the developers. According to D2.1, the state of the practice survey, we identify the following aspects of configuration that we want to amplify in STAMP:

- Software deployments
- External software and service dependencies
- Flexible architectures and scaling
- Resources

As the first step, we choose the first two aspects, i.e., the software deployments and the external dependencies as the target of the experimental amplification. In particular, the two aspects are achieved by the generation of docker images and docker compositions, respectively.

The configuration amplification is essentially a constraint solving approach. The amplifier takes as input a partial model, as defined in Section 7.1, together with a set of constraints. From the partial model, the amplifier searches for all the possible ways to complete the partial model into valid configuration models which satisfy the provided constraints.

In the remaining of this section, we briefly report the constraint solving approach for the generation of docker images and docker compositions, respectively. In each part, we will describe the specific input, the constraints, and the sample outputs.

The amplification is still in the stage of proof-of-concept experiments rather than mature tools. Therefore, the report is mainly based on a number of specific examples extracted from the STAMP usage cases.

Amplification of deployment environments

The deployment environment of a subject software contains all the platforms that the subject requires needs to deploy on. Such platforms include the operating system, the language environment, the application server, the library, the auxiliary tools, etc. Take the XWiki product as an example, the deployment environment should include a mainstream operating system (recent versions of Windows, or a distribution of linux), the Java virtual machine and a JEE service container (Tomcat, Jetty, etc.). To run test cases, it also needs maven. All these platforms or tools has to be deployed properly on the same machine (either a physical node or a virtual machine), before the target system can be deployed. The objective of the deployment amplifier is to generate a number of docker images, each of which contains the adequate and valid stack of platforms so that target system can be properly deployed into it. In the same time, each image should contain a unique stack with at least one platform that is not covered by other images.

The input and output models all confirm to the configuration meta-model described in Section 7.1. The difference is that the input describes all the building blocks whereas the output are a set of concrete images built up from these blocks.

The table below shows a very simple example.

DownloadImages			
Name	**Features**		
Java7	java7, alpine		
Ubuntu	ubuntu		
BuildRules			
Name	**Require features**	**Add features**	
PythonAlpine	alpine	python	
PythonUbuntu	ubuntu	python	
Java8Ubuntu	ubuntu	java8	
Tomcat7Java	java	tomcat7	
Tomcat8Java	java	tomcat8	
JettyJava	java	jetty	
BuildImages			
Name	from	using	features
Image0	-	-	-
Image1	-	-	-
Image2	-	-	-

The DownloadImages defines a set of images that we can already utilize, together with the features each image carries. The Java7 images provides both the feature of java version 7 and a linux distribution named

alpine. The Ubuntu images contains the latest version of Ubuntu distribution of linux. We also defined 5 different BuildRules, which, by stacking on top of another image, can introduce new features such as python, java version 8 and two different versions of tomcat. Finally, we provide three blank build images. For each of these images, we did not assign from which image it will be built, using what rules, and carrying what features. These values will be decided by the constraint solver based on a searching approach.

The constraint solver will search the potential complete models based on the following meta-model-level constraints.

```
1:  BuildImage.forall(bi1, And(
        bi1.using.requires.forall(
            f1, bi1['from'].features.exists(
                f2, Or(f2==f1, f2.allsup.contains(f1))
            )
        ),
        isunionf(bi1.features, bi1['from'].features, bi1.using.adds)
    )),
2:  BuildImage.forall(bi1, Not(bi1['from'] == bi1)),
3:  BuildImage.forall(bi1, bi1.features.exists(f1,
        Not(bi1['from'].features.contains(f1)))),
        Image.forall(e1, (e1.features * e1.features).forall(
        [f1, f2], Or(f1 == f2, Not(f1.root == f2.root)))
    ))
```

The first constraint specifies that for any BuildImage, its base image must satisfies all the features required by the building rule, and the new image will include the features added by the rule. The second constraint specifies that no image can be built on itself. The third constraint specifies that a BuildImage cannot using a building rule that does not introduce any new feature.

In this example, we want to use the constraint solver to synthesis new images that combines an application server and the Python language support. Therefore, we add a new constraint, requiring that the top image of a resulted model must contains the two features: appserver and python, or any of their sub features.

```
4:  And([wanted.features.exists(f1, Or(f1 == f, f1.allsup.contains(f))) for f in
[appserver, python])
```

Now the constraint solver is ready to search for the first result. However, the object is not to find one complete model, but several of them, so that the results together could cover a wide scope of features. As a result, instead of a one-off constraint solving, we ask the solver to perform a multi-time optimization, with the following utility function:

```
5:  solver.maximize(wanted.features.filter(f1, And([Not(f1 == fea) for fea in covered])).count())
```

Here `covered` is a list that record all the features that has already been covered by the previous results. The new result should maximize the number of features that have not been covered so far.

The following list shows three resulted complete models, after three rounds of constraint-based optimization. The first result indicates that from a standard Java7 image, we can apply the PythonAlpine rule and then the Tomcat8Java rule, and obtains an image that carries four features. The following two results are similar, but with different rules applied and carries different results. Actually, within the example model and constraints, the optimization stops after the third round: all the features has already been covered.

```
-
  image2: {from: image0, using: Tomcat8Java}
  image0: {from: Java7, using: PythonAlpine}
  covered: [tomcat8, python, java7, alpine]
-
```

```
  image0: {from: image1, using: PythonUbuntu}
  image1: {from: image2, using: Tomcat7}
  image2: {from: Ubuntu, using: Java8Ubuntu}
  covered: [tomcat8, python, java7, alpine, tomcat7, ubuntu, java8]
-
  image2: {from: image0, using: JettyJava}
  image0: {from: image1, using: Java8Ubuntu}
  image1: {from: Ubuntu, using: PythonUbuntu}
  covered: [tomcat8, python, java7, alpine, tomcat7, ubuntu, java8, jetty]
```

Amplification of external dependencies

The amplification of external dependencies follows the similar idea as the one for deployments: It takes as input an incomplete model, which is an abstraction of the docker compose model. The output include several complete models, each of which satisfies the constraints and carries different features. The table below list the sample input to the constraint solver.

Images		
name	**features**	**Depends on features**
Be_MySql	be_mysql	mysql
Be_Postgres	be_postgres	postgres
MySql5	mysql5	
MySql8	mysql8	
Postgres	postgres	
Services			
name	Image feature	image	depends_on
srv_be	be	-	-
srv_db	db	-	-

The first part of the input model includes the candidate images. These images can be the standard ones downloaded from the docker hub libraries, or the ones generated by the image amplifier. The first two images are for the business logic. The two images are specifically configured so that they can use MySql and Postgres as database, respectively. This database dependency is specified by the "dependsOnFeatures" attribute. Take the first image as an example, this dependency means that any container from this image must depend on another container that carries the mysql feature. We provide also three standard database images, two different versions of mysql and one postgres. The second part of the input model provides two services. The first service needs a business logic image while the second one needs a database image. The image and depends_on attributes of these services are empty.

The constraints are simple. First, any service must have an image which satisfies the features required by the service. Second, a service must depends on another service, and the letter carries the features that satisfies the dependent features required by the image of the former service. This is the constraint we described in the previous paragraph.

```
1:  Service.forall(s1, s1.imgfeature.forall(
        f1, s1.image.features.exists(f2, eq_or_child(f2, f1))
    ))
2:  Service.forall(s1, s1.image.dep.forall(
        f1, s1.dependson.exists(s2, s2.image.features.exists(f2, eq_or_child(f2, f1))))),
```

The constraint solving is also an iterated optimization process. The solver firsts find an arbitrary model that satisfies the constraints. After that, it tries to find new ones that bring in maximal number of new features. We omit the utility function here for the sake of simplicity.

A result can be seen below

```
-
  svr_be: {image: Be_Postgres, depends_on: srv_db}
  svr_db: {image: Postgres}
  covered: [be_postgres, postgres]
-
  svr_be: {image: Be_MySql, depends_on: srv_db}
  svr_db: {image: MySql5}
  covered: [be_postgres, postgres, be_mysql, mysql5]
-
  svr_be: {image: Be_MySql, depends_on: srv_db}
  svr_db: {image: MySql8}
  covered: [be_postgres, postgres, be_mysql, mysql5, mysql8]
```

The result contains three different complete models. The first result specifies a composition that uses postgres while the following two are the configurations that use two different versions of MySQL as databases.

9.   Conclusion

This document reports our effort on providing the fundamental building blocks for the automatic amplification of configuration testing. In particular, we report the tool to support the automatic execution of testing tasks on multiple configurations, and initial experiments on the automatic generation of mutated configurations. On the second part, we focus more on the abstract modelling of configuration specifications. The amplification is currently focused on the deployment environment and external dependencies, and next step will be introducing more aspects into the automatic generation approach.

10.   Appendix. OZ3Py: A model-based SMT constraint solver

This paper presents a Domain-Specific Language (DSL) named OZ3Py (Object Z3 in Python, pronounced as /o'zepi/) to bridge the gap between OO model and SMT constraint solving. Developers can use OZ3Py to write constraints on an OO model, using a syntax similar to OCL. In the meantime, these constraints can be used directly as input to the Z3 SMT constraint solver, and therefore enjoys the full expression power and solving features of SMT.

We implement OZ3Py as an internal DSL in Python, based on Z3's own Python API. Any constraint written in OZ3Py is also a valid Python expression, and can be used directly inside a piece of Python source code. In this way, developers can specify constraints and utilize the solver in a programmable way. On the one hand, developers can leverage the advanced Python features to help define constraints in a concise and structural way, e.g., using a function to extract a repeating part from a long constraint, or using a loop to define a series of similar constraints. On the other hand, developers can embed the invocation to the Z3 solver into long Python code to achieve complicated analysis logic based on the solving results.

OZ3Py is inspired by Alloy which brings OO paradigm into SAT. Based on more advanced SMT theories, OZ3Py has a stronger expression power than Alloy, e.g., to express constraints directly on numeric attributes of model elements. The OCL-like syntax also requires a gentle learning curve for developers. Moreover, with Z3 solver at the backend, OZ3Py supports more usage of the constraints, such weak constraints and optimization, etc.

The following figure summarizes the syntax of OZ3Py on the definition of constraints. The sample constraints we have shown in Section 8.3 are written in this syntax.

```
1  Expr ::= ObjExpr | SetExpr
2     | EnumExpr | NumExpr | BoolExpr
3  ObjExpr ::= $OBJECT | $OBJ_CONST | Undefined
4     | ObjExpr.$single_ref
5  SetExpr ::= $CLASS.allinst()
6     | ObjExpr.$mult_attr | ObjExpr[$mult_ref]
7     | SetExpr.filter(Vars, BoolExpr)
8     | SetExpr.map(Vars, Expr | [⟨Expr ⟩*])
9     | SetExpr ⟨+ | - | ^ | *⟩ SetExpr
10 NumExpr ::= $NUM_LITERAL | $NUM_CONST
11    | ObjExpr.$num_attr | SetExpr.count()
12    | SetExpr.sum(VarDecl, NumExpr)
13    | NumExpr ⟨+ | - | * | / | %⟩ NumExpr
14 BoolExpr ::= True | False | $BOOL_CONST
15    | ObjExpr[$bool_attr]
16    | ObjExpr == ObjExpr | ObjExpr.alive()
17    | ObjExpr.isinst($CLASS)
18    | ObjExpr.sametype(ObjExpr)
19    | SetExpr.contains(Expr)
20    | SetExpr == ⟨SetExpr | [⟨Expr ⟩*]⟩
21    | SetExpr.forall(Vars, BoolExpr)
22    | SetExpr.exists(Vars, BoolExpr)
23    | NumExpr ⟨> | < | == | != | >= | <=⟩ NumExpr
24    | Not(BoolExpr)
25    | ⟨And | Or | Implies⟩(BoolExpr, BoolExpr)
26 ConstDecl ::= $CONST = Const($TYPE, $str_name)
27 VarDecl ::= $VAR = Var($TYPE, $str_name)
28 Vars ::= $VAR | [$VAR ⟨, $VAR⟩*]

              Syntax sugars:

29 $CLASS.opr(Vars, Expr) ⇔
30   $CLASS.allinstances().opr(Vars, Expr)
31   where opr is filter | map | forall | exists
32 $CLASS.$ref == Expr ⇔ $CLASS.allinst()
33   .forall($CONST, $CONST[$ref]==Expr)
34 $CLASS * $CLASS ⇔
35   $CLASS.allinst() * $CLASS.allinst()
36 ObjExpr.$ref ⇔ ObjExpr['$ref']
```

The semantics of the OZ3Py language is determined by 1) how we map an OO model into a SMT problem, and 2) how we use the standard First Order Logic expressions on the SMT problem to interpret the OZ3Py expressions.

The SMT problem corresponding to the OO model constitutes of uninterpreted functions. All the functions are defined on five domains: N;Z;B are inherited from Z3, and represent the complete set of integer, real and boolean values, respectively. In addition, we introduce T as the complete set of classes defined in the metamodel, and O as the complete set of objects (both the fixed ones and the candidates) defined in the model. The two types are defined as enumerations in Z3. It is worth noting that the size O is fixed, which represents the maximal number of objects that can appear in a resulted model. Based on the five types, we first define a number of built-in functions. Function *abstract* checks whether a class is abstract, and super defines the inheritance relation between classes. The interpretation of these two functions are determined by the metamodel, we will come back to it later. Function *alive* checks whether an object is included in the resulted model, and type defines the direct type of an object stub. It is worth noting that the interpretation of function type and alive are all free, and therefore O only determines the maximal number of objects in the resulted model, without any hint on how many object a model should have, and what types they are. We also define an auxiliary function inst to check whether an object is instance of a class, considering inheritance. These functions are generic to the metamodels defined for different use cases.

Every constraint written in OZ3Py on top of a model will be instantly converted into an equivalent FOL formula on top of the SMT problem. Such a formula is composed by numeric and logic operators, and quantifiers

We implement OZ3Py as a Python module. One can use OZ3Py to define object-oriented constraints in any Python program or in the interactive Python console as long as the module is imported. The module has two parts: The model (and metamodel) definition and the constraint expressions. The metamodel definition follows the MOF standard, and comprise three Python classes for types, attributes and references, respectively. Each type wraps an enumeration item in T, and each attribute or reference wraps an uninterpreted function. A model consists of objects, their attribute values and their reference to other objects. Before calling Z3 prover, developers provide an initial incomplete model with a set of objects together with some existing attribute and reference values. They can also provide some extra objects and set them to be not alive. These objects are the seeds for the solver when it needs to consider more objects.

The constraint solver will try to search for a complete model that satisfies all the constraints. The result will be either a sample complete model or a claim that no such model can be found, which means that the constraints are conflicting.