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

Title:	WP2 – D2.3 – Enhanced prototype of the configuration amplification and report on the performance
Date:	August 28, 2018
Writer:	SINTEF, Tellu, XWiki
Reviewers:	ATOS, INRIA

Table Of Content

1. Executive Summary

This deliverable summarizes the effort in WP2 in the period of M12-M20. This deliverable is based on the previous deliverable (D2.2) and presents enhancements of the configuration testing tools developed in WP2. The document provides a description of the configuration amplification tool (CAMP) developed by SINTEF as well as progress and status on applying configuration testing techniques from other partners. In this document, we also include practical information on installation and usage of the configuration testing tools. We report also preliminary results on the task started in M12 on configuration assessment and selection.

2. Revision History

Date	Version	Author	Comments
11.06.2018	0.01	Anatoly Vasilevskiy (SINTEF)	Draft of the outline
19.06.2018	0.1	Anatoly Vasilevskiy (SINTEF)	New outline according to the online discussion
20.06.2018	0.2	Anatoly Vasilevskiy (SINTEF)	Section 6.3.1 and introduction to 6.3
05.07.2018	0.3	Hui Song (SINTEF)	Initial content for Section 6
16.07.2018	0.4	Vincent Massol (XWiki)	Added section 6.2 for XWiki and 6.2.2
20.07.2018	0.5	Anatoly Vasilevskiy (SINTEF)	All sections, content and formatting
23.07.2018	0.6	Lars Thomas Boye (Tellu)	Section 6.3.3
23.07.2018	0.7	Anatoly Vasilevskiy (SINTEF)	Harmonizing all sections
26.07.2018	1.0	Anatoly Vasilevskiy (SINTEF)	Ready for review
21.08.2018	1.1	Anatoly Vasilevskiy (SINTEF)	Revised version after the first review
21.08.2018	1.2	Vincent Massol (XWiki)	Revised section 7 and addressed comments

21.08.2018	1.3	Hui Song (SINTEF)	Revised section 6.1.1
22.08.2018	2.0	Anatoly Vasilevskiy (SINTEF)	Ready for the second review
23.08.2018	2.1	Anatoly Vasilevskiy (SINTEF)	Revision after the second review
06.09.2018	2.2	Caroline Landry (INRIA)	Adding project references, correcting typos, renaming section 10 into Bibliography

3. References

[0] Public link to the most recent version of this document: d23_config_testing_enhanced_prototype.pdf

[1] Project reference: Grant Agreement-731529-STAMP.pdf

[2] CAMP sources: https://github.com/STAMP-project/camp

4. Acronyms

MDE	Model-Driven Engineering
SMT	Satisfiability Modulo Theory
CAMP	Configuration AMPlification Tool
SUT	System Under Test
CTF	Configuration Testing Framework

5. Introduction

5.1. Purpose and Scope

The objective of this deliverable is to elaborate on the techniques and tools for configuration testing developed in the context of WP2. In the document, we provide instructions to use the tools and demonstrate their functionality on a running example.

In the deliverable, we present the configuration amplification tool (CAMP). The tool provides consistent way to manage variations of the environment of a system under test (SUT), generate executable configurations and execute generated configurations. We recap the meta-model of CAMP initially presented in D2.2 along with improvements. We describe various parts of the tool from synthesizing abstract configurations models to realizing concrete executable configurations, i.e. docker files. We also recall the functionality to execute generated configurations initially presented in D2.2 as the Configuration Testing Framework (CTF). We present a performance report to evaluate the scalability of the tool. This performance report does not try to validate the CAMP tool, rather gives an impression on a performance of the tool and potential benefits to an interested reader. Further, XWiki reports on their experiments and tools on configuration testing conducted in

the context of the project. These experiments are not focus of this deliverable. However, we decide to document them because they may result in new functionality in the CAMP tool and give a valuable insight.

We also report on an initial work with configuration assessment and selection criteria. We describe the configuration selection practices and discuss the current criteria used in XWiki and Tellu to guide the work on configuration testing.

5.2. Structure

The rest of the deliverable is organized as follows. We describe the CAMP tool in Section 6. In Subsection 6.1, we elaborate on an architecture of the CAMP tool. The subsection includes the description of different modules of CAMP. We conclude this section with Subsection 6.2 with the installation and usage instructions along with the performance report of the CAMP tool. Further, we present XWiki activities with configuration testing in Section 7. This includes experiments with CAMP and other experimentations conducted by XWiki. In Section 8, we briefly describe known configuration selection practices. Thereafter, we present criteria to select configurations currently used by XWiki and Tellu. We conclude the deliverable with a short conclusion in Section 9.

6. Configuration AMPlification (CAMP) tool

The section presents an enhanced prototype of the configuration testing tool called CAMP and recaps some of our findings from D2.2. CAMP is a configuration amplification tool to manage and maintain various configurations, generate configurations from a single configuration. CAMP also provides mechanisms to spawn off the generated configurations with a system and exercise the system on the generated configurations. Figure 1 presents a big picture of CAMP.

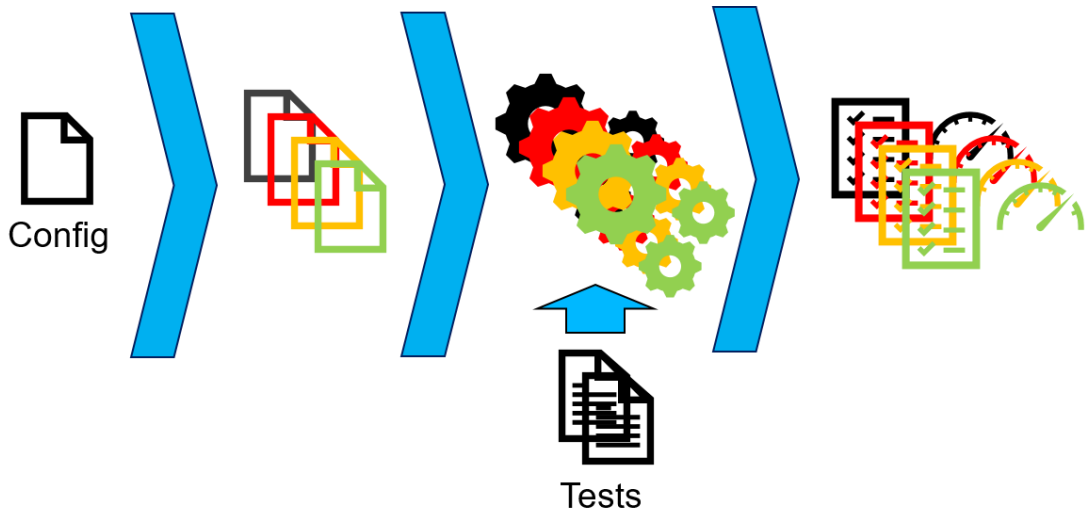

Figure 1 - CAMP big picture

In Section 6.1, we present the architecture of CAMP. Further, we reiterate the meta-model initially presented in D2.2 along with the modifications and elaborate on the process of finding configurations on a running example in Section 6.1.1. We present the CAMP realization module, that is a realization engine to execute arbitrary modifications of configuration files in Section 6.1.2. We describe briefly a module which allows

running generated configurations also referenced as the configuration testing framework (CTF) in D2.2. We describe the installation and usage procedures of CAMP and conclude with the performance report of CAMP in Section 6.2.

6.1. Architecture

Figure 2 presents the architecture of the CAMP tool. The tool consists of three modules, i.e. the CAMP configuration solver, CAMP realization and CAMP configuration executor modules.

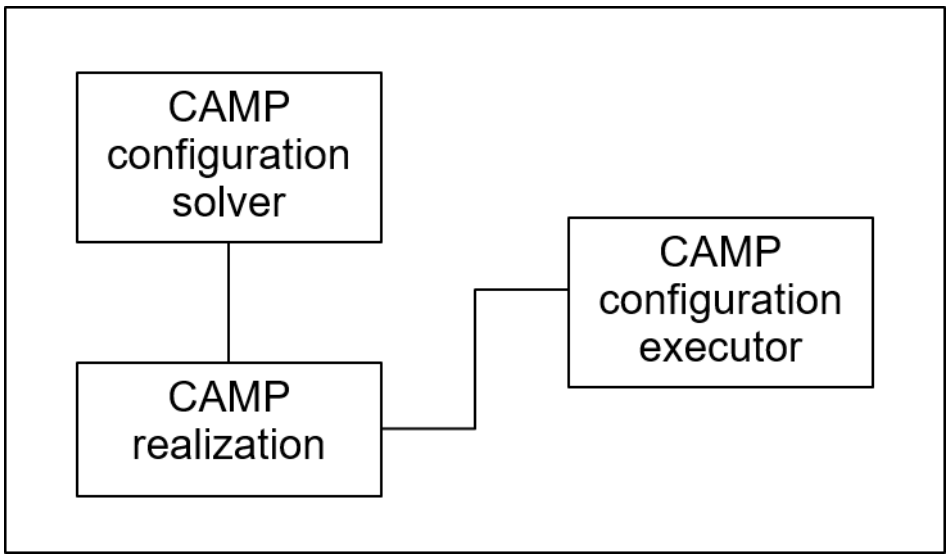

Figure 2 - CAMP module architecture

The CAMP configuration solver formalizes the CAMP meta-model and language to define abstract configurations. It also implements tooling to find new configurations based on specified models and constraints. The CAMP realization module is complimentary to the CAMP configuration solver and implements features to support the configuration solver to perform arbitrary modifications of generated configuration files. The CAMP configuration executor takes the generated configurations and can execute the generated configurations to perform testing of an application. We further elaborate on each of the modules in the subsequent subsections.

6.1.1. CAMP configuration solver

Abstract configuration model

The CAMP meta-model as shown in Figure 3 defines the concepts of testing configurations, and is currently focused on the deployment environment, the external dependencies and parameters of the software under testing. The figure is drawn in a simplified UML notation. A box represents a class and an arrow represents a reference. A multi-valued reference is annotated by an asterisk following the reference name. Any reference without an asterisk or a question mark (which means optional) is a single-valued mandatory reference.

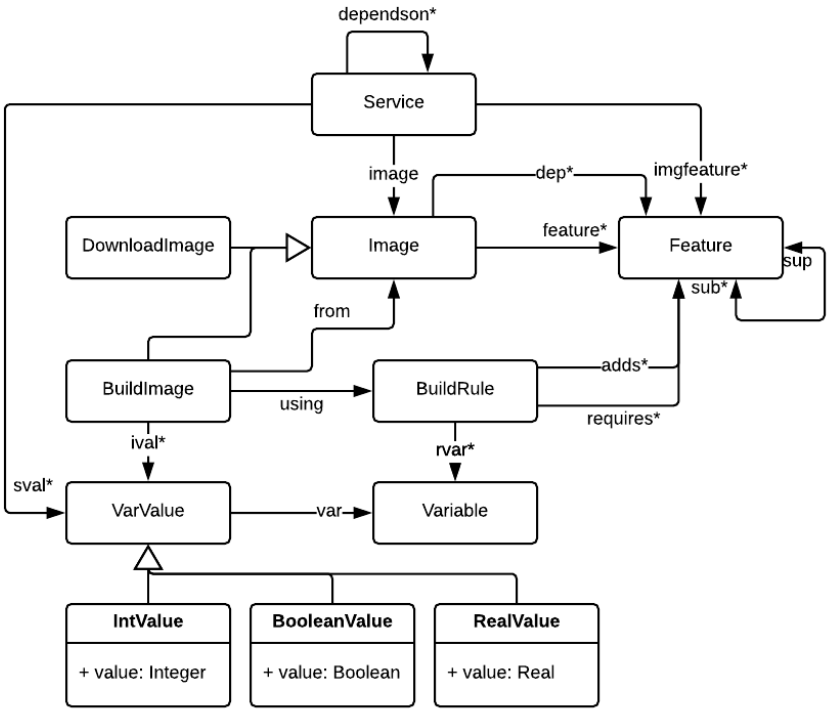

Figure 3 - CAMP meta-model

The central concept is *Service*, which represents a running piece in a testing procedure, such as a virtual machine, a container or a simple OS process. In the CAMP approach, we focus on the Docker technical stack, and therefore a service is simply a running instance of a Docker container. A service may depend on other services. A typical example of this relationship is that a test suite service depends on the service of the SUT, see Figure 4. Each service is a running instance of an image, which corresponds to a Docker Image in CAMP. An image can be an off-the-shell component, such as the standard images (DownloadImage) hosted on an image register, such as DockerHub[1]. We can also build images (BuildImage) specifically for the testing procedure.

We introduce a simplified feature model into CAMP to tag the different images. A feature may have several sub features, and all features form a forest, i.e., an undirected graph composed by several trees. An image can carry multiple features. A building block requires specific features from the base images and introduces new features to the resulted image. An image can claim to depend on a feature, which means that its instance cannot run alone, but must depends on another service whose image has the correspondent feature.

We also introduce the variable-value concepts to support the parameters. Each variable can be defined together with several alternative values. An image or a service can claim to carry a variable, and therefore it

[1] https://hub.docker.com/_/java/ all the standard Java Images registered on Docker Hub.

must contain one of the alternative values of this variable. An image can claim a variable, and the built image using this block must resolve a value for this variable.

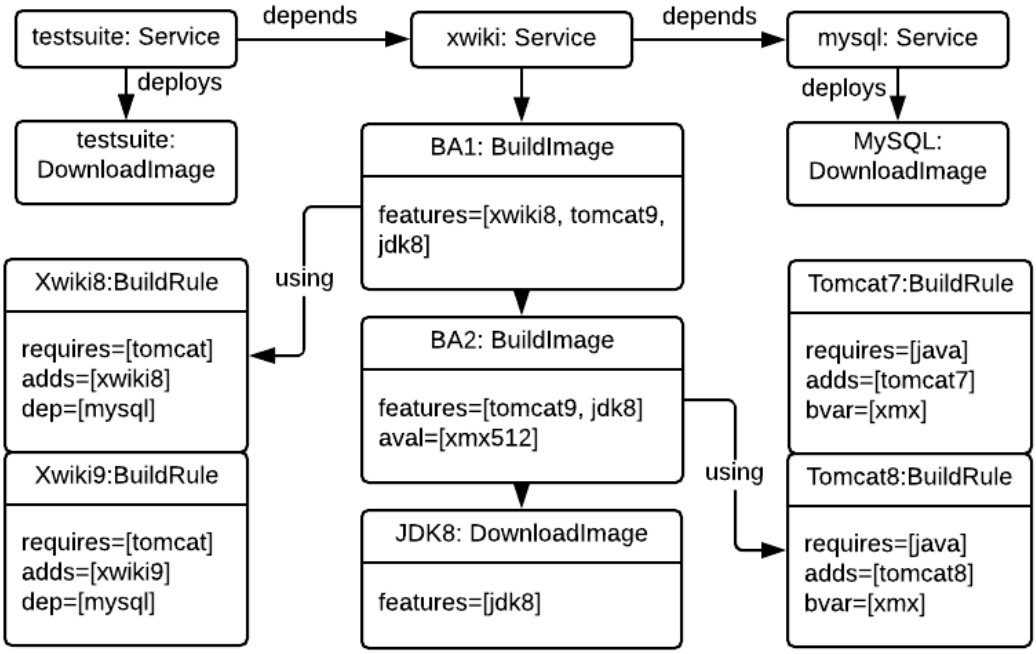

Figure 4 - A sample model of an XWiki configuration

Figure 4 gives a sample model (instance of the CAMP metamodel) of a testing configuration of XWiki. There are three running services during the testing procedure: a test suite tests an XWiki service, which in turn depends on a MySQL database. The test suite and the database are instances of off-the-shell images. For XWiki, the specific image for this configuration is built by deploying XWiki version 9 on top of Tomcat version 8. The latter is in turn deployed on OpenJDK version 8. The Tomcat building block has a variable about the size of heap, and in this configuration, it is set to 512 MB.

For a given testing scenario, part of the configuration model is fixed. In Figure 4, there must be a *testsuite* service, an XWiki service and a database service. The available download images and images are fixed, together with their features. On the other hand, the other elements of the model are flexible: There are different ways to build the XWiki image based on the provided download JDK images and the XWiki and Tomcat building blocks. There are different options for the variable, and the XWiki Service can choose different databases. The model in Figure 2 shows one of possible configuration models constrained by the fixed model parts. The objective of CAMP is to find additional compatible models.

The meta-model indicates a set of built-in constraints. All the mandatory references require a none-empty value, e.g, any service must have an image, and any built image must have a base image and uses a building rule. For the two references that are opposite to each other, the values must be consistent, e.g., if a

feature *f1* is a super feature (*sup*) of a feature *f2*, then *f1* is included in the sub feature list of *f2* (i.e., *f2.sub*). These general constraints are automatically generated by OZ3Py[2].

Along with the meta-model, we define domain-specific constraints to make sure the model instances represent valid configurations. The sample code below shows some typical constraints, written in the OZ3Py language.

```
BuildImage.forall(b, And(
  b.using.requires.forall(f1, b.from.features.exists(f2, fmatch(f1, f2)))
  isunion(b.features, bi1.from.features, bi1.using.adds),
  b.features.exists(f1, Not(b.from.features.exists(f2, fmatch(f1, f2)))
)),
BuildImage.forall(b, And(
  b.using.rvar.forall(v, b.ival.exists(vv, vv.variable == v)),
  b.ival.forall(vv, b.ival.forall(vv2,
    Or(vv == vv2, vv.variable != vv2.variable))),
  b.ival.forall(vv, b.using.rvar.contains(vv.variable))
))
Service.forall(s, And(
    s.imgfeature.forall(f1, s1.image.features.exists(f2, fmatch(f1, f2)),
  s.image.dep.forall(f1,
    s.depends.exists(s2, s2.image.features.exists(f2, fmatch(f1, f2)))),
  Not(s.dependson.contains(s1))
))
```

The first constraint block regulates the building of images based on features. The three clauses apply to all the elements that are in type of BuildImage. First of all, each built image *b* is built on a base image (*b.from*) using a building block (*using*), and the base image must satisfy all the features required by the block. Here *fmatch* is an auxiliary python function which returns an expression checking whether the first feature is equal to, or is an ancestor of, the second feather. At the same time, the resulted built image carries all the features from the base image, plus the added features of the building block. Here *isunion* is also an auxiliary function that checks if the first set is the union of the subsequent sets. Finally, a built image should carry at least one

[2] https://github.com/STAMP-project/ozepy

features that is not already provided in the base image. This prevents the conflicts such as re-installing JDK into an image that already has a version of JDK installed.

The second group of constraints are about the variables. The first two clauses specify that for any variable *v* declared in the building block used by a built image *b*, there should be one and only one variable value *vv* in *b*, and *vv* is an option under the variable *v*. In the other direction, the built image should not contain any values that does not have a corresponding variable declared in the used building block.

The third group regulates the services. If a service is declared to require a feature, this feature must be matched by the image of the service. In addition, if the image of a service is declared to depend on a feature, the service must be linked to another service whose image carries this feature. Finally, a service can never depend on itself.

In addition to the pre-defined constraints, we also support users to define their own application-specific constraints, using the OZ3Py language. The language definition can be found in Stamp Deliverable D2.2. The constraints can be written in the same input file with the base model. When working on the XWiki example, we noticed that early version of TomCat (before 8.5) does not work on Java 9, and therefore we need to introduce a constraint to avoid such invalid composition. Such application-specific constraints are not embedded in the CAMP source code, as it only applies to the XWiki use case, or similar ones that involve TomCat. Users can specify these constraints in the OZ3Py language, as part of the input file which we will discuss in the next section.

Searching for valid configurations

The OZ3Py takes as input the meta-model, the constraints, and the fixed part of the model, and searches for diverse model instances.

```
#----Meta-model definition---#

BuildImage = DefineClass('BuildImage', super_=Image)

from = BuildImage.Reference('from', type=Image, mandatory=True)

#----Feed solver with constraints---#

solver = z3.Optimize()

solver.add(genmetaconstraints())

solver.add(BuildImage.forall(...))

#----Partial model, specific to the Xwiki use case----#

Xwiki8 = DefineObject('Xwiki8', DownloadImage)

solver.add(Xwiki8.requires == [tomcat])

BA1 = DefineObject('BA1', BuildImage, suspended=true)

#----Solve constraints, specific to the searching strategy----#

if solver.check()==z3.SAT:
```

```
printmodel(solver.model())
```

The list above illustrates the very basic and simplest way of searching for a configuration model. The example is written in Python with the OZ3Py internal DSL. The first part defines the CAMP meta-models, and second part creates a constraint solver instance, and feeds it with the generated constraint and the CAMP-specific constraints. The first two parts, i.e., "meta-model definition" and "feed solver with constraints", are independent to either the XWiki use case and the searching strategy and are hard coded in the CAMP source code.

The next part defines the partial model. We define the fixed objects, such as an *BuildImage* named *XWiki8*. The fixed values of these objects, such as the *Xwiki8.requires*, are defined as additional constraints to the solver. Along with the partial model, we also define some free objects, or ``stubs'', such as *BA1*. Unlike the fixed objects that are enforced to be included to the resulted model, the free objects may or may not be in the resulted model, depending on the solver.

The last part is to search for a valid configuration. We invoke the *check* method of the solver and print out a sample model found by the solver. The solver constructs a complete model by keeping the fixed objects and their assigned reference and attribute values and deciding whether to include each free object or not, and how to assign values to all the undefined references and attributes. This part is specific to the searching strategy.

This basic searching, or constraint solving, strategy finds a random valid configuration in theory. However, its internal searching algorithm determines that the solver always find the same configuration, or one of the few configurations. As for the next step, we will provide advanced searching strategies to find diverse testing configurations, depending on the different diversification objectives.

6.1.2. CAMP realization module

To perform arbitrary modifications of configuration files, we have developed a module called CAMP-realize. The module allows specifying changes to perform on software artifacts, e.g. docker images, docker-compose files. The module is built to support the variable-value concept in CAMP described in Subsection 6.1.1. Each pair of variable and value in the CAMP meta-model maps to a pair of variable and value in the camp realization module. This defines how to realize a given pair of variable and value. CAMP finds a value to a variable that satisfies its constraints. Hence, the realization module defines how to implement the value and variable in docker image files or docker-compose files. Figure 5 shows a simplified meta-model for the CAMP realization module.

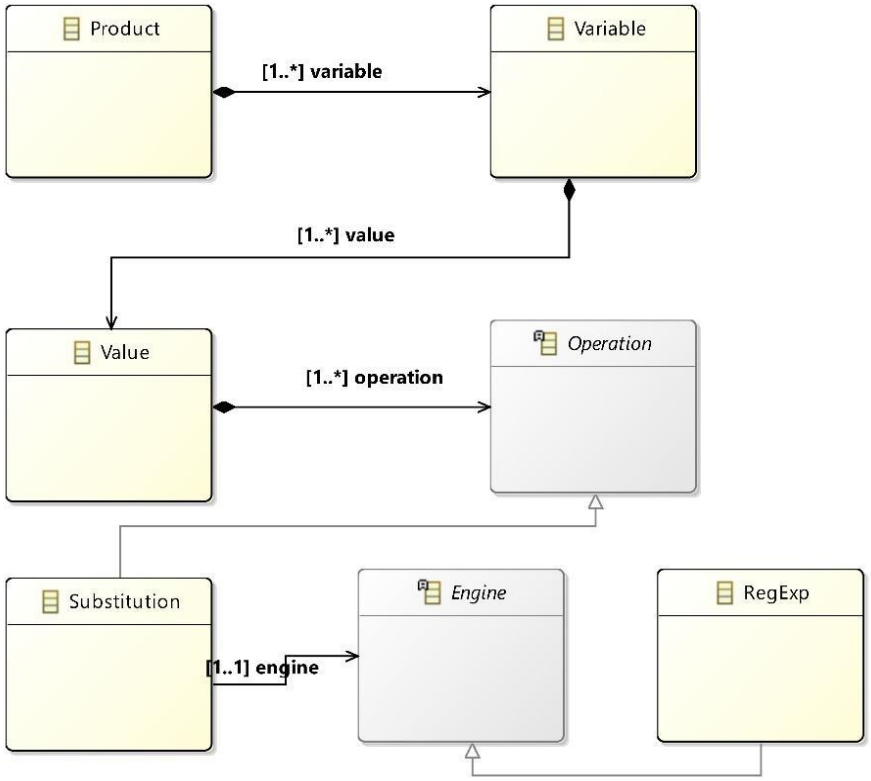

Figure 5 - Meta-model for the CAMP realization module

The meta-model has several concepts. *Product* is a concept that specifies a set of variables and their values to realize a configuration. Each *variable* may have several values to realize. Different possible values define a configuration space for the variable. Each *value* is assigned to the variable by the CAMP constraint solver. A value consists of operations to execute to realize the variable with the given value. An *operation* is an abstract type. We can have various operations. The current meta-model defines the substitution operation. A *substitution* is a simple operation which defines how to substitute one object for another. Each substitution defines a placement object, that is an object to substitute, and a replacement object, that is an object to use for the substitution. A substitution can be executed differently. This defines a type of the substitution. In Figure 4, we highlight the substitution *engine* which allows specifying placement and replacement as a regular expression.

Each concept has a concrete syntax in the YAML format. A *product model* can look as follows:

```
products:
  - product1:
      product_dir: "tests/resources/simple_e2e_regexp/tmp/product1/"
      realization:
```

```
        path: "tests/resources/simple_e2e_regexp/tmp/test_realmodel.yaml"
        variables:
          - variable1: value1
          - variable2: value1
  - product2:
      product_dir: "tests/resources/simple_e2e_regexp/tmp/product2/"
      realization:
        path: "tests/resources/simple_e2e_regexp/tmp/test_realmodel.yaml"
        variables:
          - variable1: value1
```

This product model instructs the realization module to implement two configurations, i.e. *product1* and *product2*. A location for the configuration *product1* is set by the *product_dir* attribute. This path points to the directory where we can find docker files which represent the configuration. The variables section contains a list of variables and their values they realize to yield a new configuration. For example, *product1* needs to realize two variables. We define all variables and their values in another yaml file. A location of this yaml file with all variables and possible values is defined in the *path* attribute of the *realization* section of the product also called the variable model. A variable model looks as follows:

```
variable1:
    value1:
        type: int
        value: 128
        operations:
            - substituion1:
                engine: regexp
                filename: "docker-compose.yml"
                placement: "ThreadLimit=64"
                replacement: "ThreadLimit=128"
variable2:
    value1:
        operations:
```

```
        - substituion2:
            engine: regexp
            filename: "images/Dockerfile"
            placement: "USER jenkins"
            replacement: ""
```

This variable model contains two variables. Each variable has one value; however, we can define several possible values. A value must have the *operations* section. The section describes how to realize a selected value for the variable. To realize *value1* of *variable1*, we need to replace the string *"ThreadLimit=64"* with the string *"ThreadLimit=128"* in *docker-compose.yml.* In the given example, *value1* also defines type and value, this information is needed by the constraint solver to calculate constraints. However, these attributes are not strictly required if we do not want to reason about replaced values.

To sum up, the camp realization module defines the meta-model that allows specifying arbitrary modifications of the software artifacts. We define the concrete syntax to realize the defined meta-model in two files. We need to specify a product model, that is, to specify a list of variables with their values to realize a new configuration. We also need to specify how to realize all variables with their values in a variable model. The CAMP tool can automatically calculate a list of variables and values and realize them to generate a new configuration.

6.1.3. CAMP configuration executor

CAMP configuration executor is a module to spawn off generated configurations with an application. Figure 6 sketches a main objective of the executor.

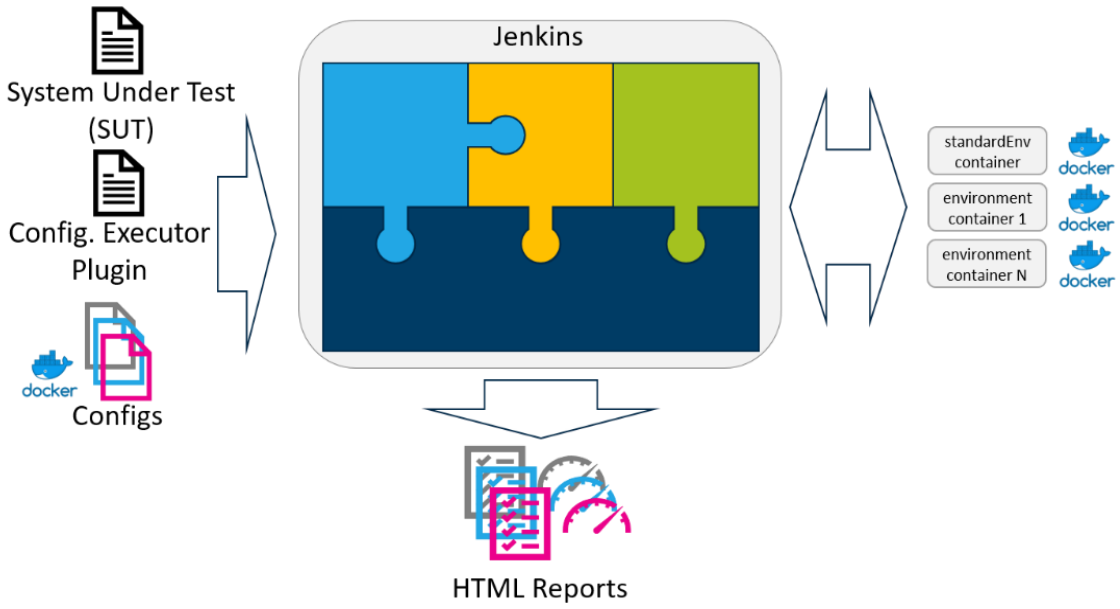

Figure 6 - CAMP executor module

It consists of several plugins which could be hooked up together to spawn off configurations, execute test cases and build a report with execution of the test cases. We can use CAMP configuration executor to run in a CI pipeline or standalone. We refer an interested reader to D2.2 in the section for configuration testing framework for more details.

6.2. Installation and performance

6.2.1. Installation and usage guide

The CAMP tool is delivered as open-source project and is available at GitHub. To get started with the tool, we need to clone the repository from GitHub.

```
git clone https://github.com/STAMP-project/camp
```

Further, we need to build a docker container which creates a platform independent package which we can run on any POSIX OS.

```
cd camp/docker/

docker build -t camp-tool:latest .
```

This creates the docker image *camp-tool:latest* with the tool. Further, we can run CAMP on an example as follows:

```
cd camp/samples/stamp/xwiki

docker run -it -v $(pwd):/root/workingdir camp-tool:latest /bin/bash start.sh
```

This command runs CAMP on the XWiki use-case and generates four configurations in form of docker and docker-compose files. A more detailed README is available at GitHub[3] with the installation instructions and other examples.

6.2.2. Preliminary performance report

We have conducted measurements to identify performance of the CAMP tool. All experiments are conducted on the following machine:

CPU	AMD Ryzen 7 1700 Eight-Core Processor
Network	RTL8111/8168/8411 PCI Express Gigabit Ethernet Controller
Storage	NVMe SSD Controller SM961/PM961
Memory	64GiB System Memory

[3] CAMP repository https://github.com/STAMP-project/camp

OS	Linux maxcloudnode5 4.13.13-sintef #2 SMP Fri Jan 5 13:13:22 CET 2018 x86_64 x86_64 x86_64 GNU/Linux

We use the XWiki use-case in all our experiments. CAMP manages and generates various configurations and helps to spawn off your application in the generated environment. Therefore, we have conducted two experiments. The first experiment aims to evaluate how fast we can generate configurations and the second experiment identifies how fast CAMP can execute the generated configurations.

In the first experiment, we conduct ten series increasing the number of features. In each series, we fix the number of features and generate configurations. We measure time to generate configurations depending on the number of features. Each series consists of ten repetitions. We calculate a median and standard deviation for each series.

In the second experiment, we conduct ten series increasing the number of configurations. In each series, we set fixed the number of configurations and execute them. We execute all configurations for the fixed number of configurations in parallel. For each configuration, we build an application, spawn off the application and execute test cases. Thereafter, the configuration is killed. Further, we measure time to complete execution of all generated configurations. Each series consists of ten repetitions. Finally, we calculate a median and standard deviation.

A primary goal of the first experiment is twofold, i.e. 1) identify how fast we can generate configurations depending on the number of features 2) identify a time progression depending of the number of features. We use the following feature model in the experiment:

```
java:

  openjdk: [openjdk9, openjdk8]

appsrv:

  tomcat: [tomcat7, tomcat8, tomcat85, tomcat9]

db:

  mysql: [mysql8, mysql5]

  postgres: [postgres9, postgres10]

xwiki: [xwiki9mysql, xwiki9postgres, xwiki8mysql, xwiki8postgres]
```

In our experiment, we generate four configurations whereas the algorithm tries to maximize the number of features in each configuration. Further, we increase the number of features and generate four configurations and measure the time to generate configurations. Table 1 in Appendix contains all made measurements. Figure 7 plots the medians and deviations from Table 1.

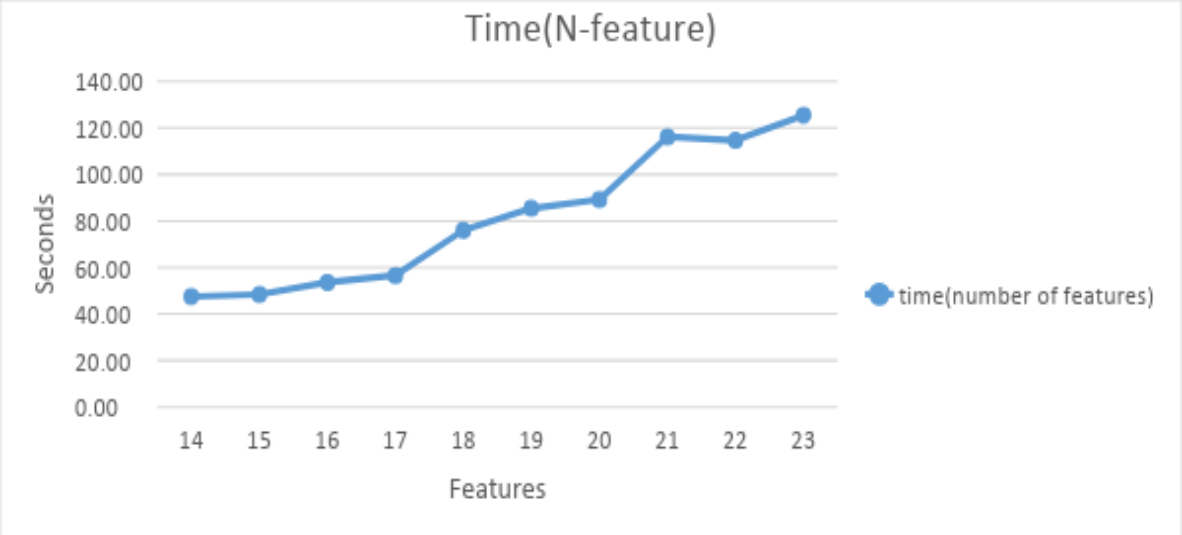

Figure 7 - Time to generate configurations depending on the number of features (median and deviation)

We can clearly see that time grows slowly. Eventually, we will face the feature interaction problem, that is adding a new feature results in a rapid increase of time to browse through a configuration space to find a candidate configuration. However, we can also observe, that we can still generate configurations in reasonable time for the XWiki use-case.

In the second experiment, we estimate performance of the CAMP configuration executor. Figure 8 shows our measurements from Table 2 in Appendix.

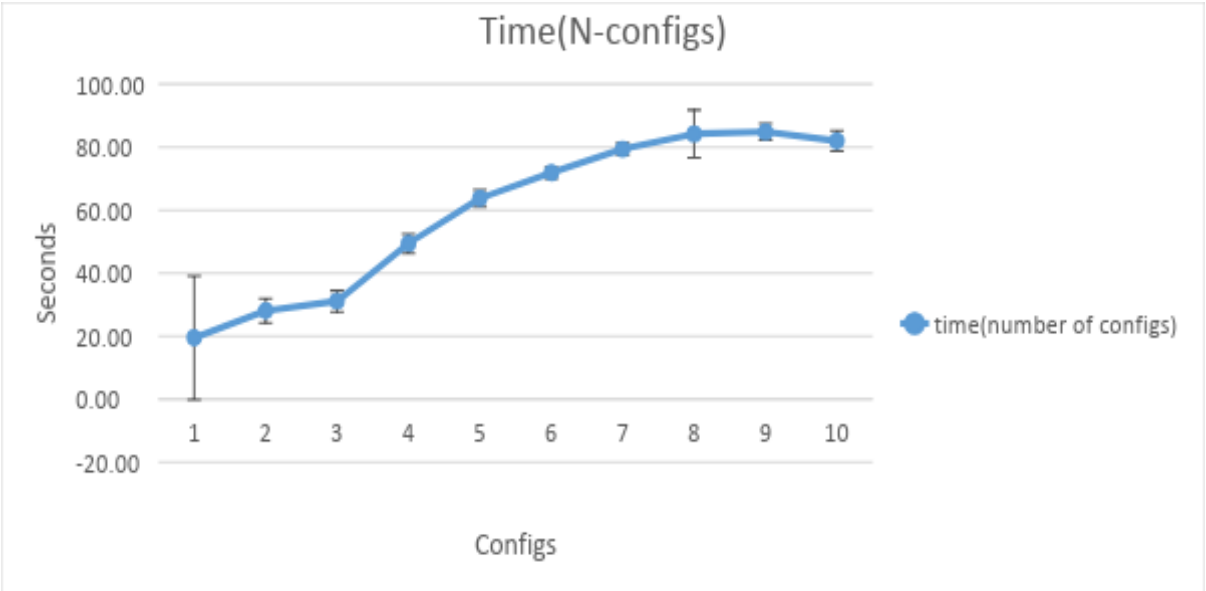

Figure 8 - Time to execute configurations depending on the number of configurations (median and deviation)

We can see that time grows linearly, and it is proportional to the number of configurations. We can also clearly see that we do not execute N as much configurations per the same unit of time.

Testing under various configuration includes various stages. Each configuration is a docker container. A docker container is an isolated unit that packages our application together with a test environment. To execute testing, a docker container needs to set up the testing environment. A set up includes downloading and installation of various software artifacts by maven in our use-case. Thereafter, maven builds the system to test and executes test cases. In our use-case, the downloading and installation phase creates a lot of I/O operations (which are slow comparing to in-memory operations) and loads the network. By executing all containers simultaneously, we increase the number of I/O operations reaching a threshold. This slows down the test containers and time to execute all test cases simultaneously grows, see Figure 8. However, we can still expect a gain from the parallel execution of the generated configurations comparing to a sequential execution. We expect that the effect grows with a reduction of the number of I/O operations.

7. Configuration testing environment at XWiki

On the XWiki project, we've been trying to define the best solution for doing functional testing in multiple environments (different DBs[4], Servlet containers, browsers[5]). So far, we've been testing XWiki automatically on a single environment (Firefox, Jetty, HSQLDB). We perform other environment tests manually[6] on a best effort basis meaning. There is no yet tooling or procedure in place to perform configuration testing systematically.

We've been conducting several experimentations, finding out issues and limitations with them and progressing towards a solution for the XWiki use case. Here are the use cases that we want to support ideally:

- UC1: Fast to start XWiki in a given environment/configuration
- UC2: Solution must be usable both for running the functional tests and for distributing XWiki
- UC3: Be able to execute tests both on CI and locally on developer's machines
- UC4: Be able to debug functional tests easily locally
- UC5: Support the following configuration options (i.e we can test with variations of those and different versions): OS, Servlet container, Database, Clustering, Office Server, external SOLR, Browser
- UC6: Choose a solution that's as fast as possible for functional test executions

Note that the experiments described below correspond to an unfinished Proof of Concept (POC), whose technologies could be possibly adopted by other industrial partners if they are deemed successful enough. They might also be eventually promoted as an alternative or complementary module to the CAMP execution engine.

[4] https://dev.xwiki.org/xwiki/bin/view/Community/DatabaseSupportStrategy

[5] https://dev.xwiki.org/xwiki/bin/view/Community/BrowserSupportStrategy

[6] http://test.xwiki.org/xwiki/bin/view/Main/WebHome

7.1. Experimentation 1: CAMP

We worked with the CAMP development team to test CAMP with XWiki and we got results showing executions of some XWiki tests using various configuration mutations showing environments in which the tests were failing and some in which they were passing. We further analyzed how CAMP works and how it could be used for XWiki and found that for the moment, it's not fulfilling several of our use cases.

Limitations of using CAMP for the XWiki use case needs:

- Relies on a service. This service can be installed on a server on premises too but that means more infrastructure to maintain for the CI subsystem. Would be better if integrated directly in Jenkins for example.
- Cannot easily run on the developer machine which is important so that devs can test what they develop on various environments and so that they can debug reported issues on various environments. This fails at least UC3 and UC4.
- Even though mutation of configuration is an interesting concept, it's not a use-case for XWiki which has several well-defined configurations that are supported. It's true that it could be interesting by running it with fixed topologies and only vary versions of servers (DB version, Servlet Container version and Java version - We don't need to vary Browser versions since we support only the latest version), but we think the added value vs the infrastructure cost might not be that interesting for us. However, it could still be interesting for example by randomizing the mutated configurations and only running tests on one such configuration per day to reduce the need of having too many agents and leaving them free for the other jobs.

7.2. Experimentation 2: Docker on CI

The main idea for this experiment was to use a Jenkins Pipeline with the Jenkins Plugin for Docker[7]. We describe the set up in D.2.2 in details. An interested reader can also refer to the XWiki blog[8] for more details. We have identified the following limitations.

- This strategy is similar to experimentation 1 with CAMP. We rely on the CI to execute the tests and doesn't allow developers to test and reproduce issues on their local machines. This fails at least UC3 and UC4.

7.3. Experimentation 3: Maven build with Fabric8

The next idea was to implement the logic in the Maven build so that it could be executed on developer machines. We found and tested the very nice Fabric8 Maven plugin[9] and came up with the following architecture in Figure 9.

[7] https://wiki.jenkins.io/display/JENKINS/Docker+Plugin

[8] https://massol.myxwiki.org/xwiki/bin/view/Blog/DockerJenkinsConfigurationTesting

[9] https://maven.fabric8.io/

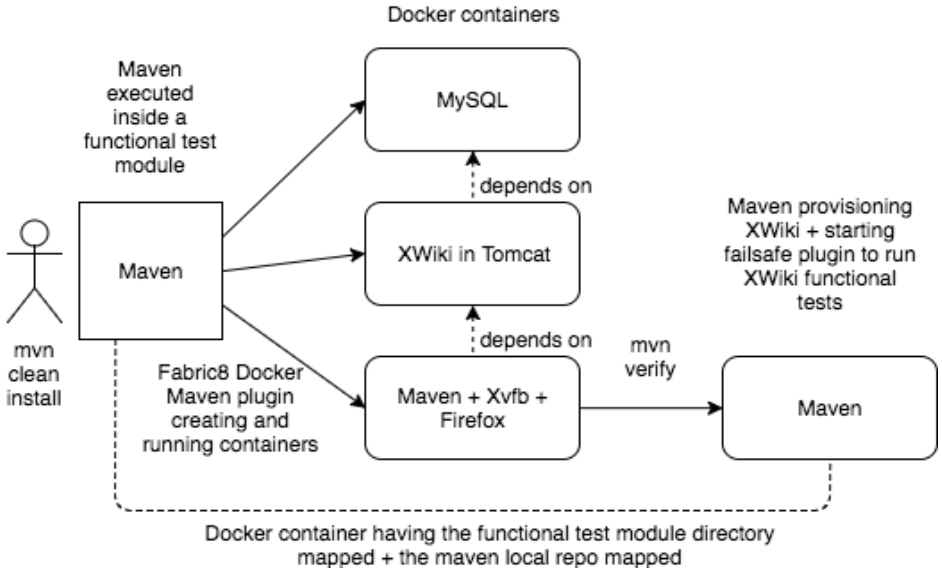

Figure 9 - The architecture of experiment 3

The main ideas:

- Generate the various Docker images we need (the Servlet Container one and the one containing Maven and the Browsers) using Fabric8 in a Maven module. For example, to generate the Docker image containing Tomcat and XWiki. Example pom.xml file in
- Listing 1 in Appendix. All the Dockerfile and ancillary files required to generate the image are in src/main/docker/*.
- Then in the test modules, start the Docker containers from the generated Docker images in Listing 2 found in Appendix. Check the full POM[10]
- Notice that last container we start, i.e. xwiki-maven is configured to map the current Maven source as a directory inside the Docker container and it starts Maven inside the container to run the functional tests using the docker-maven Maven profile[11].

Limitations:

- The environment setup is done from the build (Maven), which means that the developer needs to start it before executing the test from his IDE. This can cause frictions in the developer workflow.

[10] https://github.com/xwiki/xwiki-platform/blob/024eebd985fe6592bf048db30664c4a5d164b23f/xwiki-platform-core/xwiki-platform-administration/xwiki-platform-administration-test/xwiki-platform-administration-test-tests/pom.xml#L150-L335

[11] https://github.com/xwiki/xwiki-platform/blob/024eebd985fe6592bf048db30664c4a5d164b23f/xwiki-platform-core/xwiki-platform-administration/xwiki-platform-administration-test/xwiki-platform-administration-test-tests/pom.xml#L348-L406

- We found issues when running Docker inside Docker and Maven inside Maven, specifically when having Maven start the docker container containing the browsers, itself starting a Maven build which starts the browser and then the tests. This resulted in the Maven build slowing down and cringing to a halt. This was probably due to the fact that Docker will use up a lot of memory by default and we would need to control all processes (Maven, Surefire, Docker, etc) and control very precisely the memory they use. Java10 would help but we're not using it yet and we're currently stuck on Java8.

7.4. Experimentation 4: In Java Tests using Selenium Jupiter

The idea was to use Selenium Jupiter[12] to automatically start/stop the various Browsers to be used by Selenium directly from the JUnit5 tests.

Note that XWiki has a custom test framework on top of Selenium, with a class named TestUtil providing various APIs to help set up tests. Thus, we need to make this class available to the test too by injecting it as test method parameter for example. Thus, we developed a JUnit5 Extension named XWikiDockerExtension[13] that initializes the XWiki testing framework and that does this injection. Here's how a very simple test look like when using this JUnit5 Extension:

```
@ExtendWith(XWikiDockerExtension.class)

public class SeleniumTest

{

    @BeforeAll

    static void setup()

    {

        // TODO: move to the pom

        SeleniumJupiter.config().setVnc(true);

        SeleniumJupiter.config().setRecording(true);

        SeleniumJupiter.config().useSurefireOutputFolder();

        SeleniumJupiter.config().takeScreenshotAsPng();

        SeleniumJupiter.config().setDefaultBrowser("firefox-in-docker");

    }
```

[12] https://bonigarcia.github.io/selenium-jupiter/

[13] https://gist.github.com/vmassol/ff6a1f3a19061fb70fb025df3c04c737

```
    @Test
    public void test(WebDriver driver, TestUtils setup)
    {
        driver.get("http://xwiki.org");
        assertThat(driver.getTitle(), containsString("XWiki - The Advanced Open
Source Enterprise and Application Wiki"));
        driver.findElement(By.linkText("XWiki's concept")).click();
    }
}
```

Limitations:

- Works great for spawning Browser containers but doesn't support other types of containers such as DBs or Servlet Containers. Would need to implement the creation and start of them in a custom manner which is a lot of work.

7.5. Experimentation 5: in Java Tests using TestContainers

This idea builds on the Selenium Jupiter idea but using a different library, called TestContainers[14]. It's the same idea but it's more generic since TestContainers allows creating all sorts of Docker containers (Selenium containers, DB containers, custom containers).

[14] https://www.testcontainers.org/

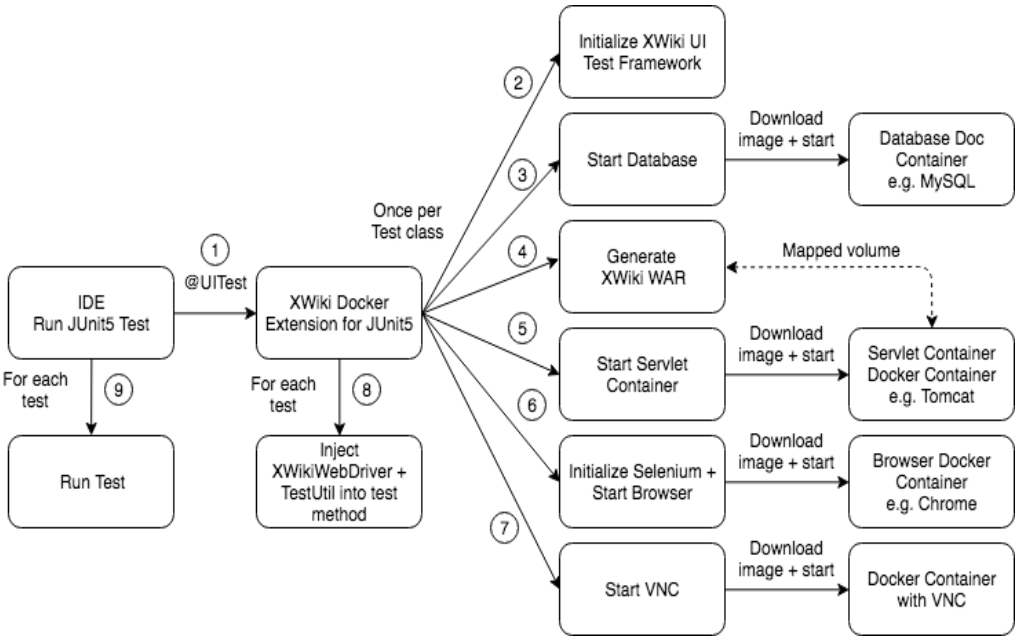

Figure 10 - The architecture of experimentation 5

The status of the approach is in Figure 10:

- Browser containers are working, and we can test in both Firefox and Chrome. Currently XWiki is started the old way, i.e. by using the XWiki Maven Package Plugin which generates a full XWiki distribution based on Jetty and HSQLDB.
- We have implemented the ability to fully generate an XWiki WAR directly from the tests (using the ShrinkWrapp library[15]), which was the prerequisite for being able to deploy XWiki in a Servlet Container running in a Docker container and to start/stop it.
- Work in progress:
    - Support an existing running XWiki and in this case don't generate the WAR and don't start/stop the DB and Servlet Container Docker containers.
    - Implement the start/stop of the DB Container (MySQL and PostgreSQL to start with) from within the test using TestContainer's existing MySQLContainer and PostgresSQL containers.
    - Implement the start/stop of the Servlet Container (Tomcat to start with) from within the test using TestContainer's GenericContainer feature.

Note that most of the implementation is generic and can be easily reused and ported to software other than XWiki.

Limitations:

- Only supports 2 browsers: Firefox and Chrome. More will come. However, it's going to be very hard to support browsers requiring Windows (IE11, Edge) or Mac OSX (Safari). Preliminary work is in progress in TestContainers but it's unlikely to result in any usable solution anytime soon.

[15] https://github.com/shrinkwrap/resolver

- Note that using TestContainers forced us to allow using Selenium 3.x while all our tests are currently on Selenium 2.x. Thus, we implemented a solution to have the 2 versions run side by side and we modified our Page Objects to use reflection and call the right Selenium API depending on the version. Luckily there aren't too many places where the Selenium API has changed from 2.x to 3.x. Our goal is now to write new functional UI tests in Selenium 3 with the new TestContainer-based testing framework and progressively migrate tests using Selenium 2.x to this new framework.
- The full execution of the tests takes a bit longer than what we used to have with a single environment made of HSQLDB+Jetty. Measures will be taken when the full implementation is finished to evaluate the total time it takes.

Future ideas:

- Discuss with the CAMP developers to see how their mutation engine could be executed as a Java library so that it could be integrated in the XWiki testing framework. Namely two issues were open on the CAMP issue tracker to discuss this:
  - Be able to run configuration testing on developer machines and as part of tests[16]
  - Be able to use the configuration mutation algorithm as a library[17]

7.6. Experimentation summary

At this point in time we're happy with the last experiment (experimentation 5). It allows running environment tests directly from your IDE with no other prerequisite than having Docker installed on your machine. This means it also works from Maven or from any CI. We need to finish the implementation, and this will give the XWiki project the ability to run tests on various combinations of configurations.

Once this is done we should be able to tackle the next step which involves more exotic configurations such as running XWiki in a cluster, configuring a LibreOffice server to test importing office documents in the XWiki, and even configuring an external SOLR instance. However, once the whole framework is in place, I don't expect this to cause any special problems.

Last, but not least, once we get this ability to execute various configurations, it'll be interesting to use a configuration mutation engine such as the one provided by CAMP to test various configurations in our CI. Since testing lots of them would be very costly in term of number of Agents required and CPU power, one idea is to have a job that executes, say, once per day with a random configuration selected and that reports how the tests perform in it.

8. Configuration assessment and selection criteria

Assessment of software configurations is an important task to choose the most relevant configurations to test. In general, criteria to conduct a selection could vary depending on a goal of testing. Most of the approaches to select configurations are tackling a combinatorial explosion problem, that is the number or all configurations is too high in general. Thus, it is practically impossible to generate every possible configuration and perform testing. Therefore, those approaches trying to define a representative subset

[16] https://github.com/STAMP-project/camp/issues/12

[17] https://github.com/STAMP-project/camp/issues/13

among all possible configurations. The CAMP tool is also aiming to provide an approach to pick useful configurations to test.

In this section, we survey the most typical approaches to evaluate and pick a configuration to test. Further, we outline how the use-case providers in the project perform assessment of their software configurations and choose selection criteria to pick a candidate configuration. We will follow up this task in the upcoming deliverables to present a complete survey with devised strategies to select configurations to test.

8.1. Configuration selection practices

Software Product Lines (SPL) [1] is a relevant area in research which deals with various configurations of software and discusses questions relevant to generation of the most relevant configurations to test. Those approaches fall roughly into three main categories: 1) brute-force, 2) feature-interaction, and 3) search-based approaches.

We can generate all possible configurations and execute given test-cases against such software configurations. Further, we collect required metrics to find out a subset of the configurations which satisfies given functional or non-functional criteria. Pohl et al. [2] categories such strategy as the Brute Force Strategy (BFS). In case, a configuration space is large, this approach results in an NP-complete decision problem. That is, the time required to generate all configurations increases very quickly as the number of variations in configurations grows. Pohl et al. suggest also the Pure Application Strategy (PAS). PAS advocates to avoid generation of any configuration unless requested by anyone.

To check functionality of a software product, we can generate only such configurations, where a pair of parts in the configuration interacts with each other, so-called the 2-wise interaction, and skip others. By doing so, we reduce the overall number of configurations to test. Kuhn et al. [3] claim that we can find up to 50-70% of the bugs by testing only those configurations which cover all 2-wise interactions between the parts of the configuration. Including, 3-wise interactions into generated configurations may increase a defect ratio up to 95%. Unfortunately, the number of possible interactions also grows exponentially, which makes the task of generating such configurations computationally hard when the configuration space (the number of parts to vary) is large. Fortunately, combinatorial interaction testing (CIT) [4] approaches are emerging. For example, ICPL [5] can generate configurations from a configuration space of the industrial size.

We can also employ search-bases approached to find an optimal configuration which satisfies non-functional properties, such as performance. By doing so, we reduce the number of generated configurations and produce only those candidates to test which meet required properties. For example, Siegmund et al. [6] present an approach for non-functional optimization that allows qualitatively specify, quantitatively measure and describe non-functional properties in a feature model to support automatic derivation of new configurations to meet specified goals. Soltani et al. [7] suggest formalizing required non-functional properties, propagate them into a feature model. Further, they transform the feature model into hierarchy of task to solve to derive required configuration and employ automated planning techniques [8] to build a plan to derive a new configuration with required properties. Barner et al. [9] suggest a holistic methodology and tooling tailored to mixed-criticality systems. The methodology employs genetic algorithms to evaluate a configuration space and find several alternatives to satisfy functional and non-functional properties of a desired system.

In the following subsection, we briefly discuss how XWiki and Tellu select configurations to test their products.

8.2. Configuration selection criteria at XWiki

XWiki is a classical web application. The XWiki platform can be delivered to various customers and installed on their premises. Thus, XWiki can run in multiple environments and set up against various database systems. The XWiki team aims to ensure that their application is run properly against various configurations. Therefore, a main criterion to pick candidate configurations is a proliferation of the most popular databases, browsers, servlet containers and java versions. As touched upon in Subsection 6.2, the interesting configurations for XWiki are:

- Java 8+[18]
- Database[19]:
  - HyperSQL 2.3.3+
  - MySQL 5.x+
  - PostgreSQL 9.x+
  - Oracle 11.x+
- Browsers[20]:
  - IE11 latest only
  - Edge latest only
  - Firefox latest only
  - Chrome latest only
  - Safari latest only
- Servlet Containers:
  - Jetty latest
  - All versions of Tomcat 7.x+ except[21]
    - >= 9.0.0.M5 and < 9.0.0.M10 for the 9.0.x branch (fixed in 9.0.0.M10)
    - >= 8.5.1 and < 8.5.5 for the 8.5.x branch (fixed in 8.5.5)
    - >= 8.0.34 and < 8.0.37 for the 8.0.x branch (fixed in 8.0.37)
    - >= 7.0.70 and < 7.0.71 for the 7.0.x branch (fixed in 7.0.71)
    - 8.0.32

Those are the configurations supported by XWiki and thus the ones that XWiki needs to ensure are working. Any other are not interested to XWiki since they're not supported.

---

18 https://www.xwiki.org/xwiki/bin/view/Documentation/AdminGuide/Installation/#HHardwareandSoftwarerequirements

19 https://dev.xwiki.org/xwiki/bin/view/Community/DatabaseSupportStrategy

20 https://dev.xwiki.org/xwiki/bin/view/Community/BrowserSupportStrategy

21 https://www.xwiki.org/xwiki/bin/view/Documentation/AdminGuide/Installation/InstallationWAR/InstallationTomcat/

8.3. Configuration selection criteria at Tellu

In this subsection, we give an update on Tellu's work in WP2, with a focus on configuration assessment and selection. A comprehensive description of Tellu's work related to configuration testing is found in D5.5, together with other work related to this use case. Here is a quick recap, to give context to the rest of the section. We have worked on exploring the many different forms of configuration involved in a TelluCloud deployment, and how they relate to each other. We have also been changing some of the ways we configure the system as part of the STAMP work. We have experimented with different orchestration tools to handle deployment and see how much of the deployment we can automate and describe in configuration files. We have ended up with Kubernetes as the preferred orchestration tool, configuring how to deploy the set of dockerized service components into a cloud system with a set of yaml files. We have been developing the Tellucloud testing tool needed to run tests on queues, microservices and the complete system. This type of test consists of sending messages into the system/component under test, receiving results over various protocols, and validating the results. The tool is used for both functional, performance and stress testing. The tests themselves are now described in configuration files, which may also be a target for amplification. And we have developed the tests themselves, and integrating them in our pipeline, running some of them in our Jenkins CI.

We have started to apply the theoretical framework described in deliverable D2.1 on TelluCloud configuration. The main categories presented there are very useful to describe the forms of configuration and ultimately how amplification with STAMP tools can be applied. The categories are the capability for configuration, how configuration is done, how to test the effects of configuration and finally the enhancement. In the following table we list identified capabilities based on the different ways configuration is done. An important aspect is that of explicit (configuration file) versus implicit configuration. The latter can be in the form of the infrastructure and external services which is available. A goal of using Kubernetes has been to make much of the infrastructure and other deployment configuration explicit, putting it in a file which is available to amplification.

Capability	How to configure
Service properties: set of properties for each microservice, whatever is needed to configure its functionality. Some are common to all microservices.	Explicit through files. There is a findit.properties file which is shared by all microservices, and some have separate config.properties file.
Mediator properties: configures the mediator part of each microservice, which connects it to the other microservices. Specifies the type of queue to use and affects scaling.	Explicit through file service.properties. Has a common file for all services in a deployment, and service-specific additions/overrides.
Database properties. TelluCloud uses Hibernate, which is configured with database type and instance.	Explicit through file hibernate.cfg.xml.
Logging configuration.	Explicit through file logback.xml.

Edge setup. The edge micro-service can be set up with different edges, which represent different protocols for devices delivering data to the system. Configuration of which edges to include and their properties.	Explicit through file edge.json.
Device library. Defines the supported devices and their commands.	Explicit through a set of XML files used by multiple microservices.
Rule templates. These define the templates available for creating service logic to run in the rule engine.	The templates are stored in the database. We have a set of them as files which can be imported, but not currently any way to do so automatically.
Accounts. Tests rely on having service provider and customer accounts, with users for access and content such as assets, devices and rules.	In database. For tests we are so far partly using a pre-existing database instance and partly initializing content through the HTTP API.
Service components. Not all micro-services need be included in the system, and more importantly there can be variability in their versions. We may want to test an updated version of one service while keeping the other unchanged.	Implicit in which components we deploy.
Database: The database instance. We can use different types of SQL database.	We use both explicitly deployed database instances (docker) and Amazon's database service.
Queue/mediator: We can use different types of queues or other channels to connect the micro-services.	As with the database, this could either be a component we explicitly deploy as a docker (RabbitMQ) or a provided service (Amazon SQS).
Docker image configuration. For our own micro-services the Dockerfile is very simple and does not currently have any relevant variability, but for infrastructure components like database and queues there is relevant configuration here.	Explicit through Docker files.
Deployment configuration using Kubernetes. There are very important parameters related to performance and scaling, such as requests and limits for memory and CPU allocation.	Explicit through Kubernetes yaml files.

Of all the forms of configuration we have listed, a few are primary targets for configuration testing. The Kubernetes deployment configuration is a very interesting target. So far, we have been doing some manual variation and testing of memory and CPU allocation to the different services. This is an important point to optimize and balance, as we must make sure that each service gets enough resources to run correctly (and stay running for a long period of time) under relevant amounts of workload, while we must also make sure to

not use more resources than needed, as these are costly and/or limited. This is an area where amplification would be very useful. Related to this is also the number of instances of each service. Altogether we want to find the optimal number of instances and resources for each instance for specific workloads.

We also need to test with different databases and different queues. We currently have two or three alternatives for each for real-world use, and in addition there are also test implementations we use in some tests. This aspect may not be well-suited to automatic amplification. The change is not simply one of parameters in a configuration file, as it can either be a service we deploy as a docker (PostgreSQL and RabbitMQ) or an external service (Amazon database and queue services), so the configuration change is more complex. There is also not much variability and probably no feature interaction between queue and database, although there may be interaction with other forms of configuration. Variations of Docker image configuration through the Dockerfile is relevant for infrastructure components, which primarily means the database and queue when we don't use the external services.

Now we come to our current question of configuration assessment and selection. In some cases, it is a simple choice based on what we have selected to support (which databases and queues) or what makes sense. But we see two main themes for how assessment and selection get interesting. One issue is that of relations between the different forms of configuration. They are not always independent, and a change in one form of configuration may need changes in other configuration files. This is a question of consistency between the different forms of configuration. For instance, we configure aspects of the internal network in Kubernetes files, and this must be consistent with the addresses of connected services we specify in service and mediator property files. If we change the database this must be reflected in the Hibernate properties, and if we change the queue this must be reflected in the mediator properties. There are also dependencies between the configurations and the tests. The tests need the correct URLs to reach the exposed service APIs, and the correct rule templates and account contents present in the deployed services. With the TelluCloud test tool we have placed all test parameters which can be affected by system configuration into test configuration files, so that it is easy to modify these and make new versions to correspond to new configurations of the system under test.

The other theme is that of assessing the configuration based on test results. We do not generally have any other way to assess a configuration. Our system tests will either check functionality, in that we get the expected outputs from the system based on a specific set of inputs, or check performance, with metrics for how fast we get the outputs. A configuration is not valid if a functional test fails, so amplified configurations can be discarded in this case (we may want to run the tests several times, as system tests can easily be flaky, but this is also usually reason to discard the configuration). The performance tests are the most interesting for configuration amplification, in terms of minimizing use of resources while still staying above certain performance thresholds. We can specify the thresholds in our test setup, failing tests if we go below them. We could also use the performance metrics from tests to direct the configuration amplification. In the manual approach used so far, we have simply tried some different values for the resource settings in the Kubernetes files, until we get the system running reasonably well. For selection of configuration values, this has so far been done by the developer selecting or modifying within what seems like reasonable limits. Here the developer could explicitly specify the ranges of numerical values to vary, as input to the amplification.

9. Conclusion

In this deliverable, we have summed up the effort in the period of M12-M20. We have presented the CAMP tool, that is a tool to systematically perform configuration testing. We have introduced different modules of CAMP and elaborated on their interplay in the tool. We also give some practical information of the installation

and usage of the tool. We have assessed performance of the tool on the use-case. Further, XWiki reports on their experiments with the CAMP tool and configuration testing in XWiki in general. We conclude the deliverable with an initial work on the task on the configuration assessment and selection criteria for configuration testing in XWiki and Tellu.

10. Bibliography

1. Clements, P. and L. Northrop, *Software product lines: practices and patterns*. Vol. 3. 2002: Addison-Wesley Reading.
2. Pohl, K., G. Böckle, and F.J. van Der Linden, *Software product line engineering: foundations, principles and techniques*. 2005: Springer Science & Business Media.
3. Kuhn, D.R., D.R. Wallace, and A.M. Gallo, *Software fault interactions and implications for software testing.* IEEE transactions on software engineering, 2004. **30**(6): p. 418-421.
4. Cohen, M.B., M.B. Dwyer, and J. Shi, *Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach.* IEEE Transactions on Software Engineering, 2008. **34**(5): p. 633-650.
5. Johansen, M.F., Ø. Haugen, and F. Fleurey. *An algorithm for generating t-wise covering arrays from large feature models*. in *Proceedings of the 16th International Software Product Line Conference-Volume 1*. 2012. ACM.
6. Siegmund, N., et al., *SPL Conqueror: Toward optimization of non-functional properties in software product lines.* Software Quality Journal, 2012. **20**(3-4): p. 487-517.
7. Soltani, S., et al. *Automated planning for feature model configuration based on functional and non-functional requirements*. in *Proceedings of the 16th International Software Product Line Conference-Volume 1*. 2012. ACM.
8. Ghallab, M., D. Nau, and P. Traverso, *Automated Planning: theory and practice*. 2004: Elsevier.
9. Barner, S., et al. *Building product-lines of mixed-criticality systems*. in *Specification and Design Languages (FDL), 2016 Forum on*. 2016. IEEE.

11. Appendix

Table 1 - Time to generate configurations depending on the number of features

Features/Time											Median	Std. Dev
14	47,46	47,45	47,49	47,49	47,52	48,04	47,49	47,52	47,48	47,46	47,49	0,177263
15	48,83	48,82	48,32	48,4	48,42	48,4	48,41	48,87	48,36	48,45	48,42	0,218419
16	53,54	53,52	53,57	54,03	53,6	53,54	53,59	53,66	54,22	53,6	53,60	0,23847
17	56,71	56,63	56,55	56,57	57,1	56,62	56,62	56,63	56,56	57,09	56,63	0,209008
18	76,21	75,94	75,84	75,9	76,47	76,03	76,17	75,91	76,57	75,85	75,99	0,260404
19	85,51	85,54	85,51	85,39	85,38	85,9	85,3	85,34	85,67	85,4	85,46	0,180259
20	89,38	89,56	89,16	89,15	89,58	89,13	89,21	89,35	89,13	89,08	89,19	0,18379

21	115,9	116,28	115,91	116,35	115,92	116,13	116,39	116,33	116,41	115,9	116,21	0,228106
22	114,7	115,1	114,53	115	114,6	114,54	115	114,54	114,38	115,2	114,66	0,293038
23	125,46	125,79	125,49	126,1	125,35	125,5	125,7	125,3	125,9	125,1	125,50	0,295261

Table 2 - Time to execute configurations depending on the number of configurations

Configs/Time											Median	Std. Dev
1	79,04	14,17	13,01	14,70	16,78	19,04	20,10	20,11	20,29	19,98	19,51	19,63422
2	30,13	33,41	32,43	25,66	31,42	34,22	25,93	25,53	24,56	25,01	28,03	3,852284
3	36,23	28,56	29,28	28,90	29,14	32,46	31,74	30,50	37,68	35,25	31,12	3,337694
4	44,48	51,38	51,14	53,52	49,09	52,43	48,56	49,75	47,31	44,71	49,42	3,061089
5	56,48	61,21	62,50	63,93	63,82	64,51	63,67	60,87	65,35	65,08	63,75	2,665645
6	69,82	73,91	71,51	74,09	72,93	69,05	72,21	70,93	74,16	70,58	71,86	1,837314
7	80,63	76,85	81,17	80,00	82,27	77,39	78,06	79,26	79,42	76,69	79,34	1,898369
8	61,58	80,87	78,12	79,41	84,09	85,76	84,39	87,88	85,89	84,33	84,21	7,553298
9	84,16	83,58	87,91	87,38	85,75	86,40	85,58	81,69	80,69	81,04	84,87	2,614914
10	81,32	82,12	81,30	84,74	80,38	86,11	84,14	80,14	81,79	90,69	81,96	3,255508

Listing 1 - Fabric8 in Maven module

```
[...]
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <configuration>
    <imagePullPolicy>IfNotPresent</imagePullPolicy>
    <images>
      <image>
        <alias>xwiki</alias>
        <name>xwiki:latest</name>
        <build>
          <tags>
            <tag>${project.version}-mysql-tomcat</tag>
            <tag>${project.version}-mysql</tag>
            <tag>${project.version}</tag>
```

```
          </tags>
          <assembly>
            <name>xwiki</name>
            <targetDir>/maven</targetDir>
            <mode>dir</mode>
            <descriptor>assembly.xml</descriptor>
          </assembly>
          <dockerFileDir>.</dockerFileDir>
          <filter>@</filter>
        </build>
      </image>
    </images>
  </configuration>
</plugin>
[...]
```

Listing 2 – Generated Docker images

```
[...]
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <executions>
    <execution>
      <id>start</id>
        <phase>pre-integration-test</phase>
        <goals>
          <goal>start</goal>
        </goals>
        <configuration>
          <imagePullPolicy>IfNotPresent</imagePullPolicy>
          <showLogs>true</showLogs>
          <images>
            <image>
              <alias>mysql-xwiki</alias>
```

```
            <name>mysql:5.7</name>
            <run>
            [...]
          <image>
            <alias>xwiki</alias>
            <name>xwiki:latest</name>
            <run>
            [...]
          <image>
            <name>xwiki-maven</name>
            <run>
            [...]
              <volumes>
                <bind>
                  <volume>${project.basedir}:/usr/src/mymaven</volume>
                  <volume>${user.home}/.m2:/root/.m2</volume>
                </bind>
              </volumes>
              <workingDir>/usr/src/mymaven</workingDir>
              <cmd>
                <arg>mvn</arg>
                <arg>verify</arg>
                <arg>-Pdocker-maven</arg>
              </cmd>
    [...]
    <execution>
      <id>stop</id>
      <phase>post-integration-test</phase>
      <goals>
        <goal>stop</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```