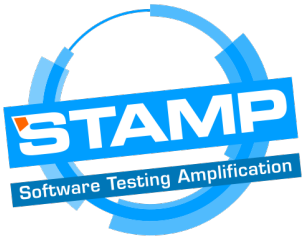


European Commission
Horizon 2020
Communications Networks, Content & Technology
Cloud & Software
Research & Innovation Action
Grant agreement n° 731529

Title:	WP2 – D2.1 – Report on the State of Practices for Configuration Testing
Date:	May 31, 2017
Writer:	SINTEF, Atos, TellU
Reviewers:	INRIA, ActiveEon

Table Of Content

1. Executive Summary

This document summarizes the work and achievement under Task 2.1: "Survey state of practice for configuration specification and automatic configuration planning". We report the survey we undertook within the STAMP industrial partners and a theoretical framework on the status of configuration test as an outcome from the survey. We also describe in detail how two partners, TellU and Atos, currently do configuration test. Based on the state of the practice, we describe the status of two potential tools for the improvement of configuration test, i.e., the Docker container technology and the research on Software Product Line testing. The state of the practice will be used as the internal reference for the following tasks in WP2.

2. Revision History

Date	Version	Author	Comments
18-Apr-2017	0.1	Hui Song (SINTEF)	Outline and draft of Section 6
05-May-2017	0.2	Brice Morin (SINTEF)	Section 7.1
07-May-2017	0.3	Hui Song (SINTEF)	Section 4, 5, 7.2
09-May-2017	0.4	Lars Thomas Boye (TellU)	Section 8.1
09-May-2017	0.5	Jesus Gorronogoitia Cruz (Atos)	Section 8.2
10-May-2017	1.0	Hui Song (SINTEF)	Sent for internal review
19-May-2017	1.1	Hui Song (SINTEF)	Revision according to internal review

3. Objectives

The objectives of this document can be summarized as follows.

- Set up a common understanding within the STAMP partners on what is configuration test, how it is performed currently, and what improvement is expected.
- Identify the primitive challenges of configuration test
- Collect the potential tools, both technical and academic ones, to address the challenges
- Provide a reference for the subsequent Workpackage 2 tasks on improving and amplifying configuration test

4. Introduction

Software systems are not only a piece of source code. To different extent, both the external environments and the internal settings influence their behaviour and performance. The former include the available resources, software platforms, depended services etc., and the latter include predefined parameters, the flexible architectures, etc. In this way, the same piece of code will run in different ways, also known as in different configurations, by different users. A thorough test of the system's functionality on only one configuration cannot guarantee that the system will run properly in all the potential configurations, and therefore, it is the developers' interest to test their software under multiple and representative configurations. We call this quality assurance (QA) activity the configuration test. The activity is usually performed after unit tests and before the release of the system. The focus is not only the functional correctness of the system, but also the quality and consistency of the system under different configurations.

Configuration test is more important for modern software systems, especially the ones under the microservices style. On the one hand, one of the main motivation behind microservices is the flexibility on configurations: the systems are easier to replace some of its components (the microservices), to reconnect between the components, and to scale out some individual components. On the other hand, tightly coupled with the cloud technology, the microservices are hosted by flexible environments, such as potentially different cloud providers and different choices of resource offers. This enlarges the configuration space of microservices systems.

Despite the increasing importance, the current practice of configuration test is far from mature, comparing to unit test. There are not standard tools that are widely adopted for configuration test, such as JUnit for unit test. Moreover, there are not a standard process on how to perform configuration test, or even a common understanding about the definition and scope of configuration test.

As the first task of WP2 on the amplification and efficient execution of configuration test, our main objective of T2.1 is to establish a common understanding among the STAMP partners about what is configuration test, how the industry partners perform configuration currently and what are their expectations, in order to provide a reference for the following WP2 tasks. We narrow down the scope into the internal STAMP partners, as well as a number of selected open source projects hosted by OW2, in order to provide a dedicated reference for our own work. However, in the same time, since the studied projects covers a wide scope of software types, from libraries, standalone products to software as a service (SaaS), we believe that the findings in this task are relevant to the general research on configuration test.

We study the state of the practice of configuration test based on an internal survey with the STAMP partners. The survey focuses on the system configuration capability (how the current systems support configuration?), the current practice of configuration test (how many different configurations are tested and how to prepare these configurations?), and the expectation on the improvement of configuration test (how many configurations should be tested in the future, and how to produce these configurations?). The survey comprise 20 questions from the above categories, and we received in total 14 different responses to the survey. Based on these answers, we also did further study on the public materials of the subject systems, such as the open source repository and the documentation. Furthermore, we met two partners for further discussion into relevant details.

The survey reveals that the current practice of configuration test is still a craft rather than engineering. In summary, most teams only test the system under very few configurations, if not only one, especially on the external environments. For the teams that do test multiple configurations, they test a small number of different one, in a low frequency, and usually only do sample simplified testing tasks on each configuration. The reason behind this the high cost of doing configuration test: On the one hand, it is expensive to prepare the system to test under different configurations, for it usually involves the deployment of complex dependencies and the setting up of the system and its resources. On the other hand, it is expensive to run testing tasks on each of the configurations. Configuration test is usually performed in the integration level, and involves UI and performance. Therefore, the tasks are more complex than unit test and often need manual effort. In summary, the current practice of configuration test within the STAMP consortium is far from systematic and automatic. The status also influences their expectation on the improvement on configuration test in the future: Most of the survey answerers only expect to test, in the future, a very small number of configurations regularly, and relatively small number of configurations (e.g., 5-10) on specific conditions, such as before major release or after patches related to configurations. We will explain our understanding about the state of practice of configuration test in the form of theoretical framework of important concepts and their relations.

The status drives us to rethink about the plan of WP2. Since the cost of performing configuration test directly determines how many configurations can be tested, we concludes that the efficient execution of configuration test is more urgent in the moment than the generation of diverse configurations. In other words, we need to first set up the tools and environment to support systematic and automatic execution of test on multiple configurations, before we can start looking at generating more configurations. Based on the two types of cost for configuration test we mentioned above, we foresee two major directions, i.e., the efficient construction and preservation of configurations and the automatic execution of testing tasks on these configurations. We will investigate such configuration test environment based on the container technology and the Docker toolset, and the existing research achievements on the testing of software product line. We will report our initial attempt and lessons learned on using Docker for testing, and give a short literature review on the state of the art of software product line testing.

As case studies of the status and the future directions, we will also report the details on configuration test with two concrete examples, i.e., the TellU Cloud and the systems from the Atos R&D department. These two examples will also be the start point of the upcoming experiments in WP2 on setting up the automatic configuration test environment. We will also discuss briefly the future plan.

As a report on the state of the practice, this document will not contain a literature survey on configuration test from the academic point of view. The main reason for this is that we want to focus on understanding the

status in the industry in order to provide solutions that directly apply to the STAMP use cases. Moreover, the scope of configuration test is too wide. For example, the mainstream research interest in the academic is how to help users to test if their configuration is valid, which is not within the scope of STAMP, which focuses on how to assist developers in testing. Last but not the least, we have already reported a brief literature survey in the Description of the Work.

The rest of the report is organised as follows. Section 2 will brief the survey process. Section 3 presents a theoretical framework to explain the status and challenges of configuration test. Section 4 follows the survey and presents the state of practice of two selected projects in detail. Section 5 introduces the potential ways to use Docker and Product Line Testing research to address the challenges. Section 6 discusses our plan and Section 7 concludes the report.

5. Internal Survey on Configuration Testing

In order to understand the state of the practice within the STAMP partners on configuration testing, we undertook a survey with all the partners. The survey is based on a pre-defined questionnaire, together with supplementary studies to the material of individual partners as well as direct meetings with selected partners.

The main objective of this survey is to build up an initial understanding about what configuration and configuration test mean to different software development teams, how these teams currently work on testing configurations and how they expect their current practices could be improved.

In this section, we will introduce how we undertook the survey, and then give a brief summary about the answers we received. The main outcome of the entire survey process, based on the analysis of the answers, the direct material and the one-to-one meetings, will be presented in the next section.

5.1. Survey process

The table below summarises the entire process of the survey, including four major activities, and across half a year. We also listed some important events during the process.

12	1	2	3	4	5
	Making questionnaire				
	Receiving responses from partners				
		Supplementary study			
				Document writing	

16/12: questionnaire sent

16/12: first response received from XWiki

03/03: working session with INRIA on initial findings (skype)

30/03: use case discussion with TellU

05/04: last response (so far) received

27/04: use case discussion with ATOS (skype)

The main assets of the survey are the questionnaire and the responses to it from the partners. The questionnaire consists 20 questions in two major categories, i.e., how the subjects do configuration testing currently and how they foresee the improvement to the current practice. The former can be also divided into the nature of configuration and the practice of configuration test. The latter comprise how people expect to improve configuration test and how many configurations they would like to test in different situations. Through the questionnaire, we would like to collect the information and opinions on the STAMP partners' understanding and expectation. Under each question, we tried to list some options based on our own understanding, just as a guidance, or a supplement to help you understand the question. We recommended

the subjects to utilise the "other" option and write down their opinions directly. We designed the questionnaire in the way that it should take approximately 10 minutes for each answerer.

#	Question	Type
1	How do you currently deploy your software for testing	Capability
2	How does your software support external variety	
3	Does your software contains internal configurable variants	
4	Is your software able to change its configuration automatically at run-time?	
5	Do you know how your users/integrators validate their configurations of your software? (e.g., do they test the configuration files they write?)	Current status
6	Do you test your software under more than one external environments or internal settings	
7	If you do test different configurations, how do you set up those configurations before testing	
8	Have you met any bugs/issues reported by users which are specific to their own configurations	
9	If you answered yes to the previous question, would you please briefly describe an example, like what was the configuration like, and what causes the "bug"	
10	Would you consider testing your software under multiple configurations?	Expectations
11	If you answered "yes" or even "maybe", what would be your motivation?	
12	How would you expect to set up multiple configurations for testing	
13	If we would introduce the container technology (docker) for configuration testing, which of the following features would be interesting to you	
14	If you have answered yes to the previous question, then how do you test your software for such adaptation capability	
15	Would you consider testing some configurations that are automatically generated from your base configurations	
16	How many different configuration would you consider to test [We would test {XXX configurations} each time we build the software]	Scale
17	How many different configuration would you consider to test [We would test {XXX configurations} before major releases]	
18	How many different configuration would you consider to test [We would test {XXX configurations} only occasionally, in dedicated sessions]	
19	How many different configuration would you consider to test [We would test {XXX configurations} only when we extend software to introduce new configuration possibilities (e.g., adding parameters to the config file, or support new cloud providers)]	
20	How many different configuration would you consider to test [We test {XXX configurations} occasionally, in a dedicated session focusing on performance, resiliance,]	

5.2. Summary of responses

In total, we received 14 responses to the survey. Among them, six are from projects hosted by OW2. The other responses are from two Engineering groups and then each from the rest of the STAMP industry partners. The software behind these responses can be categorized into the following three groups.

- Libraries. Those are the fundamental software products that are used by other software through local APIs. Such systems usually have very simple dependencies, such as Java Standard Edition and a set of internal libraries that are built together with the them. A sample from the responses is sat4j in OW2.
- Standalone products. These systems have some environment dependencies, such as JEE (Java Enterprise Edition) application servers, database, browser, etc. Product users may deploy the systems in different ways, resulting in different configurations. The majority of systems behind the survey responses belong to this category.

- Software as a Service. The system is mainly deployed on the vendor's side and are maintained by the vendor as well. Customers use the system through remote APIs. Samples are the system from ATOS, TellU and Engineering.

Configurations have different meanings in the different categories of the systems. In libraries, the configurations are usually simple, and vendors do not expect many bugs related to configurations. For standalone products, the configuration space is larger and the actual configuration is made and decided by the users. The vendor often suggest some recommended or officially supported configurations, such as specific databases and application servers, in a specific range of versions. They test the system under these configurations, and in the real cases, the configuration-related bugs are often found in the configurations outside of the recommended scope. For SaaS, the dependents are fixed at the vendor side, but they still need to test alternative choices, such as different application servers, in order to find the best one. In the same time, these systems usually support a flexible architecture, such as vertical or horizontal scaling, architecture adaptation etc., and they need to test these alternative configurations in advance.

Regarding to the current status of configuration test, the majority of the teams behind the responses do test multiple configurations, but in a rare frequency and small numbers. These configurations are usually predefined and remain stable in a quite long period. The configurations are either maintained physically or as virtual machines. They run test cases or small tasks simulating the actual usage. Some of the testing tasks running on the configurations have to be done manually, especially when it is related to the UI testing.

When asked about the expectation of configuration testing, most answerers are willing to test more configurations, even with some of them generated automatically. But they expect that the generation is under their control, within a predefined scope or reviewed manually before used for testing. In terms of actual number of configurations they would like to test, most of them expect testing a small number of configurations (around five) in a regular base, and test a little more (around 10) in specific situations, such as before major release, after changes specific to configurations and on dedicated sessions only for configuration testing.

We will not repeat the responses to each questions in this report, but instead, we will present how we understand from the survey responses in the next section in the form of a theoretical framework.

5.3. Summary

This section describes the survey we have conducted with all the industrial partners in STAMP. The survey is based on questionnaires, documents and meetings, around how the software systems are configured, how the team test configurations now and how they expect to improve the current practice. The survey gives us an firsthand material on understanding the state of the practice of configuration test.

6. A Theoretical Framework for Configuration Testing

This section presents a “theory” of configuration testing as a set of concepts that we find out as important from the surveys, as well as the relations between these concepts. The objectives of this theory are expected to be the following:

- Explain how configuration testing is typically performed by STAMP partners
- Reveal where the challenges and opportunities may exist
- Position the potential research of STAMP, with the relations to other activities
- Identify the case studies for the decided research

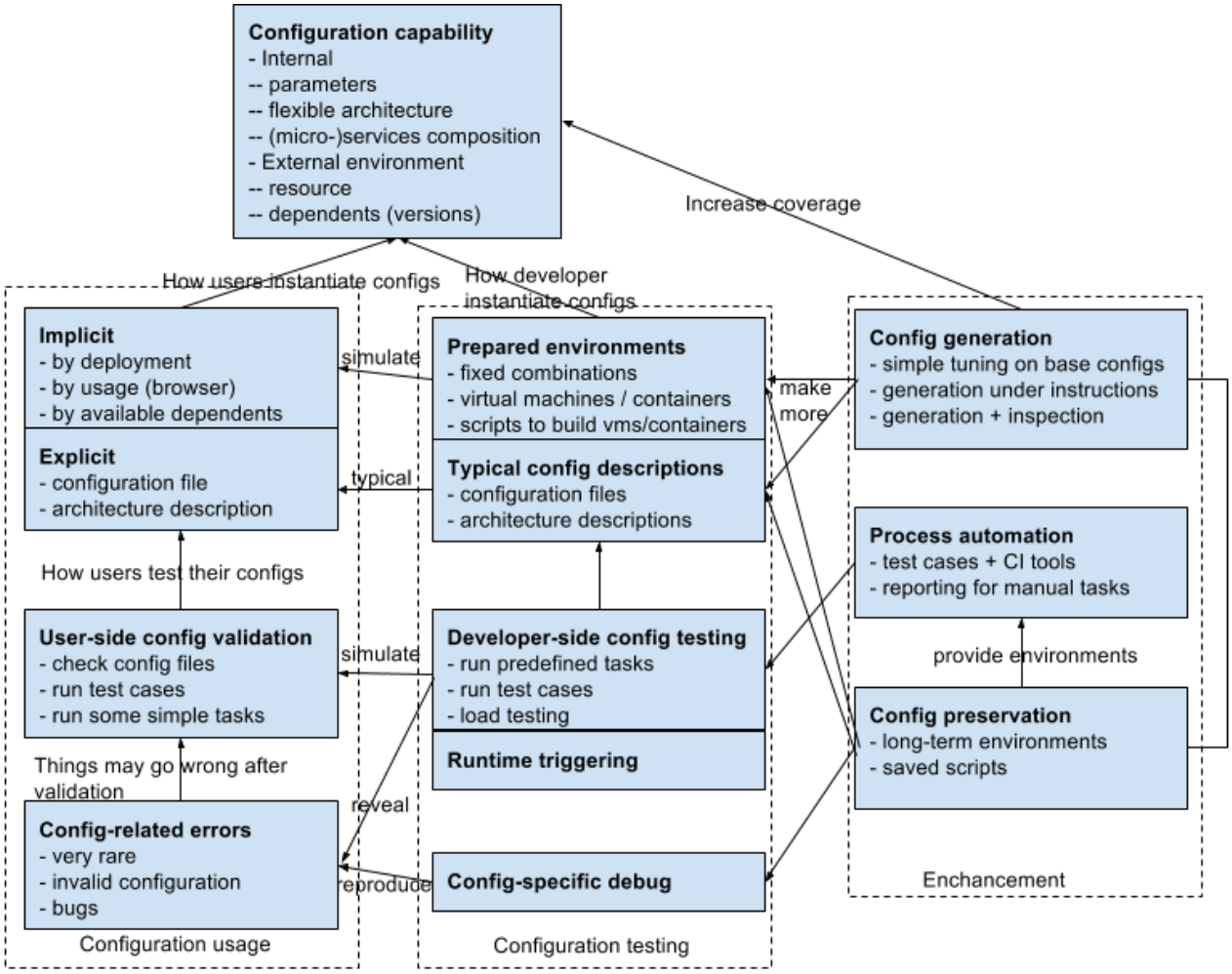
6.1. Categories

Category	Description	Stakeholder
Configuration capability	How the system is able to be configured	developer
Configuration usage	How the system is configured when usage	users
Configuration testing	How the system is tested under different configurations	developer / QA team

Testing automation and amplification	How to improve the current testing activity	STAMP

6.2. The concepts

We introduce the concepts and their relations, under each of the four categories.



Configuration capability

The system itself should be able to be configured, or “run in different ways”. This includes the internal configuration, e.g, some systems have parameters that determines the system’s behaviour, have flexible architectures so that they can scale out when necessary or replace some of its own components. In an extreme case, some systems are a loosely coupled composition of services or microservices, and the composition is changing easily. There are also external configurations, such as the hardware resources used by the system, and the dependent software platforms, services or tools. One typical example is that a couple of surveyed systems are running on JEE application servers (Imixs-workflow), and some has to be used via a browser (Xwiki). One important issue here is the version of dependent services or platforms.

Configuration usage

The system will be configured before used by users. Here we differentiate between implicit and explicit configurations, depending on whether the users need to describe the configuration by a file or not.

Implicit configuration are set by the administrators when the system is deployed (resources or platforms), by users when the system is actually used (such as which browsers is used), or by the system itself during runtime (available resources or dependent services). Explicit configuration are described by a system-

specific configuration file. An important difference here is that the configuration space is more predictable in an explicit configuration than an implicit one. For example, if the user may have to choose from one of the supported platforms when they must configure it via a configuration file, whereas if they can deploy the software to a platform without explicitly declaring, it is possible that they choose a platform that are not predicted by the developers.

After setting up the system, the users may perform a validation before actually using the system. Some systems require the users to run a set of test cases that the development team has provided, but most of the time, the users just launch the system, run a couple of simple tasks only to see if the system is able to start up with the current configuration. The development team may also place some validation function during the system launching period, to check if the system is deployed with valid dependents (for implicit configuration) or if the configuration file is valid (for explicit configuration). The latter is more common, but still hard to be thorough and sufficient.

During the actual usage, the system may have errors that only appears in the specific configuration. The reason could be either that the configuration itself is “wrong” or that the system has bugs that expose only to the particular configuration. There is not a very clear line between the two situations, because one can always say that it is the system’s fault of not tolerating a problematic configuration. The partners have given some examples to such errors, related to the version incompatibility to platforms or the unpredicted behaviour when scaling up.

Configuration testing

The developers, or the QA within the development team, usually test the potential configuration before releasing the system. The purpose is to reduce the chance of the configuration-specific errors. The activities done by the developers are correspondent to the ones by the users: They need to set up the system into some configuration, and run the system under these configurations.

The configurations set up by developers should be representative to the ones that will be actually used by the users. For explicit configuration, they need to prepare some typical configuration files or architecture descriptions, whereas for implicit ones, they need to prepare some environment that simulates the one the system will be running in. Such environments could be simply described in an instruction for QA engineers to deploy and test, or can be some prepared testing machines, virtual machines or containers. Obviously, the physical machine with all required dependents is the most efficient one in terms of testing setup: every time the system is being tested, the QA team just need to build the system on the machine. But due to the cost, there can’t be many physical environments prepared. Virtual machines and containers are cheaper. Some teams also prepare some scripts or Docker files to build the temporal environments for testing.

We can call these configurations prepared for testing a configuration pool. According to the surveyed partners, the size of this pool is usually small. Most of them test only a handful of different configurations regularly. On some particular circumstances, such as before release or after major change related to configuration, they may concentrate on testing tens of different relevant configurations.

On top of the configuration pool, the QA team need to actually run the system to test all the configurations work as expected. Theoretically, the testers should run the entire test cases on each configuration, but what they actually do is something between a thorough test and running some simple tasks. One challenge here is the manual testing. Configuration testing is usually performed in the integration testing stage, because most configurations are global ones. However, comparing to unit testing, configuration testing does not enjoy the same automation level at the moment, especially when the UI is involved. For this, what Xwiki is doing may be representative: They defined a set of simple UI testing tasks, and ask the testing team to run the tasks on different versions of the software, under different configurations.

In the survey, some partners also mentioned the debugging under particular configurations. The challenge here is to reproduce the configuration where the error occurs, in order to check whether it is the problem of the configuration or the system itself, and solve the problem.

Configuration testing enhancement

The last category of concepts are about what we as researchers could do to improve the state of practice of configuration testing. Here I foresee two major directions:

- To reduce the time and cost spent on testing one single configuration.

- To increase the coverage of the tested configurations in the entire configuration space.

In the first direction, there is quite some room for improvement, considering the current automation level. There are partners reporting that they integrate configuration testing into CI tools for automatic testing before each release. They do so by embedding the configuration pool into the test cases. This should work well for explicit configurations, but for implicit ones, it would be difficult to integrate into the test cases with executable instructions about how to construct the entire environment for testing. Another challenge in this direction is the manual testing. We probably do not want to touch how to automate the manual testing tasks themselves, such as UI testing, since this is not really the problem of configuration. However, we could try to accelerate the preparation of configurations before testing, and the reporting of testing results afterwards.

In the second direction, the state of the practice is that the testing teams keep some base configurations (explicit files or implicit environments), and tune it when necessary for specific circumstances. It seems to be a promising way to increase the coverage by generating automatically new configurations instead of asking the development team to build many ones by hand. However, in the survey, almost every surveyee tried to prevent us from “generate what we want”. Instead, they suggest us either generate new ones only using the mutators they provide us, or to do it under strict constraints. Some surveyees would also like to review the generated configurations before testing them.

One way to accelerate the preparation of testing configurations is to preserve some long-term configurations so that they are ready to be tested immediately when needed. Considering the cost to maintain such long-term configurations across different versions of the system, container technology would be a promising approach. We can reuse some pre-built images with all required environments (or even directly use some official images available online), and deploy the target version of the system into it before testing. Afterwards, the container instances can be simply thrown away. One surveyed project (SeedStack) seems to have used this way for “smoke testing” their system (https://github.com/seedstack/smoke-tests). They test their product on four JEE application servers by instantiating the official Docker images of these platforms, respectively, and deploy their product into the containers. They used the maven Docker plugin to automate the building, instantiation, deployment, testing and report.

6.3. Summary

This section presents what we have learned from the survey. In summary, the major takeaways from the survey are the following.

- Configurations have different meanings in different systems
- The external and architectural configurations that are set up implicitly (i.e., without a configuration file) are the most challenging to test
- The process of configuration testing is not mature, i.e., the processes of preparing configurations and running testing tasks on them are not systematic or automatic.
- Development teams do not test a large number of configurations, due to the big cost to prepare configurations and run testing tasks
- There are two directions to improve configuration test, i.e., to reduce the cost of testing one configuration and to increase the coverage of tested configurations. The first one appears to be more urgent based on the state of the practice.

7. Tools for Configuration Testing

From the survey, we identified two directions of expected improvement on configuration test, i.e., to reduce the cost of setting up configurations and run testing tasks on them, and to increase the coverage of the tested configurations. In the first direction, the recent development on operating system level virtualisation, or the container technology, provides a promising way to simulate the external environments of systems as well as the topology between system components, in a much cheaper and flexible way than traditional physical or virtual machines. In the first subsection, we will discuss the usage of Docker, the mainstream container environment, and the orchestration engines based on Docker, to set up configuration test environment. In the section direction, the research approaches in Software Product Line testing have accumulated some theoretical and technical tools to enlarge of the coverage of tested configurations for a target system. In the second subsection, we will give a brief literature review in this domain.

7.1. Docker orchestration for configuration testing

Over the past few years, a number of orchestration engines for containers has been released as open source. This for example includes Kubernetes[1], building on 15 years of experience at Google and now donated as open source to the Cloud Native Computing Foundation[2], but also Netflix's Conductor[3], Spotify's Helios[4], RedHat's OpenShift[5] (itself relying on Kubernetes), Mesosphere's Marathon[6], etc. Most of those approaches has first been developed internally as a strategic tool to provide more abstraction and control over a set of containers. Though they are all slightly different in practice, they offer a rather homogeneous set of abstractions:

- Services, for highly-available, long-living computational units
- Tasks, for short-lived computational units
- Load-balancing, to automatically share the incoming load (e.g. HTTP requests) to a set of instances realizing the load-balanced service
- Scheduling, to allocate services or tasks to physical or virtual nodes, depending on constraint (e.g. making sure enough resources are available on the node to deploy a service or a task).
- Scalability, to dynamically scale up or down services, by instantiating or removing replicas, which will typically load-balanced, and to continuously monitor the current replicas to maintain the desired number of replicas at all time e.g. by creating a new replica if one failed.

Such orchestration capabilities are now also directly available as part of the core Docker technology through Docker Swarm[7]. In particular, Docker Swarm does now integrates with the latest version of Docker Compose (v3), so as to allow deployments (a.k.a stacks), I.e. a set of services, to be described in declarative files and automatically deployed onto a swarm of Docker nodes.

For example, the script below describes a deployment that has been used in FP7 HEADS in order to assess the scalability of the technical solutions offered by this project.

```
version: '3'
services:
  mosquitto:
    image: 'bmorin/mosquitto'
    # deploy three replicas on big nodes (testing MQTT is a non-goal)
    deploy:
      replicas: 3
      placement:
        constraints:
          - "engine.labels.sin.cpu == intel"
  fieldnode:
    image: 'bmorin/fieldnode'
    depends_on:
      - mosquitto
    deploy:
      replicas: 256
    # Make sure gateways can run in reasonably small containers
      resources:
        limits:
          cpus: '0.02'
          memory: 64M
        reservations:
```

1 https://kubernetes.io/

2 https://www.cncf.io/

3 https://netflix.github.io/conductor/

4 https://github.com/spotify/helios

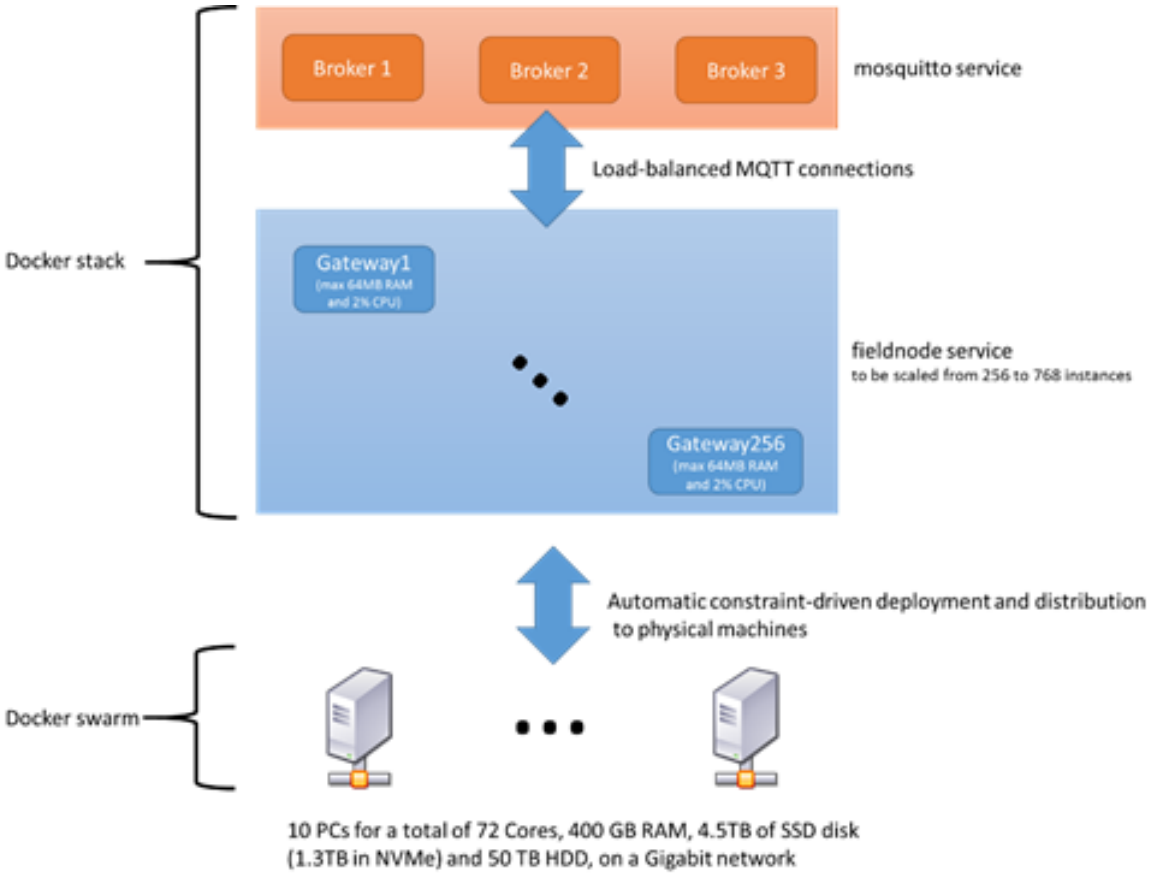
5 https://www.openshift.com/

6 https://mesosphere.github.io/marathon/

7 https://docs.docker.com/engine/swarm/#feature-highlights

```
        cpus: '0.01'
        memory: 16M
#Extend default network so that we can have more than 256 containers
networks:
  default:
    ipam:
      driver: default
      config:
        - subnet: 10.84.0.0/16
```

This script is basically equivalent to the deployment described by the Figure below:



Regarding the objectives of STAMP WP2, Docker Swarm and Docker Compose files provide a very rich framework for configuration testing and configuration testing amplification. In particular, the following leverages can be used to amplify configurations:

- The number of replicas of a service can be adjusted both statically before the actual deployment and also dynamically after the system has been deployed. This allows for testing similar configurations at different scales
- Different kinds of constraints can be applied to the deployment in order to guide or force Docker Swarm in the deployment process by suggesting or specifying a subset of the nodes where a given service can be deployed. In the script above, we specify that the mosquitto service (a MQTT Broker) should be deployed on Intel-based nodes, with the following constraint: "engine.labels.sin.cpu == intel". Constraints can also specify range of resources that each replica of a service can use. In the script above, we constrain the replicas of the fieldnode service to use at most 2% of the host's CPU and 64 MB RAM. Such constraints allow for testing a given configuration in different environments, e.g. to assess the behaviour of a configuration in case resources become limited and see if the services can scale down and cope with limited resources or whether it will crash.

Most of the container orchestration engines provide similar features and the choice of Docker Swarm and Docker Compose will not prevent to apply STAMP WP2 results to other orchestrators such as Kubernetes. One key criteria to choose Docker Swarm and Docker Compose is that those technologies are now part of the core Docker infrastructure, and hence easy to install and deploy, providing easy to replicate configuration testing environments. As needed by the use cases of the project, WP2 will also seek to apply its result on other relevant orchestrators.

A common limitation of those orchestration engines is the textual notations (typically YAML or JSON-based), and more importantly tool support around those text files, that are currently provided. This lack of proper tool-support and the lack of a proper formalization for the input files makes it tedious for developers to provide the initial set of files that could later on be amplified, as errors are typically detected by the engines when trying to deploy the system, resulting in a cost and time-consuming error-retry process. WP2 has thus developed an initial tool for parsing Docker Compose v3 files , which allow to extract an object-oriented model out of those files. This OO abstraction will facilitate tool support, analysis and transformation (amplification) of configurations.

7.2. Software Product Line Testing

In the academic testing community, a relevant branch to configuration test on the developers' side is the Software Product Line (SPL) testing. A product line is a collection of systems with a considerable amount of hardware and/or code in common. The primary motivation for structuring systems as a product line is to allow customers to have a system tailored for their purpose and needs, while still avoiding redundancy of hardware and/or code. Within a product line, customers will run the software in their own configurations (products), and it is the developers' obligation to ensure that all the potential configurations will have the required quality. From this point of view, the approach to SPL testing is useful to address the major challenges of configuration test, i.e., how to prepare multiple configurations for testing, how to run test tasks efficiently on these configurations, and how to maximize the coverage of the configuration space.

Product line testing approaches can roughly be divided into four categories: 1) approaches not utilizing product line assets, 2) model-based techniques, 3) reuse techniques based on regression testing and 4) subset heuristics.

Contra-SPL-philosophies

Pohl et al.[8] discuss two techniques for testing product lines that do not utilize product line assets. These are the Brute Force Strategy (BFS) and the Pure Application Strategy (PAS). The Brute Force Strategy (BFS) lives up to its name. The strategy is merely to produce all products of a product line and then test each of them. This strategy is infeasible for any product line but the very smallest. This is because the number of products generally grows exponentially with the number of features in a product line.

The Pure Application Strategy (PAS) is simply to not test anything before a certain product is requested. When a product is requested, this product is built and tested from scratch. Although this strategy is easy to use and feasible, it does not provide validation of products before they are requested. It would be good to somehow utilize the product line assets in order to ensure to some extent that the products work when they are requested and built. This is the basic concern of the product line testing technique discussed below.

Reusable Component Testing

Reusable Component Testing (RCT) can be used when a product line is built out of components that are composed into products. These reused components can be tested in isolation. If they function correctly, we have more confidence they will not fail because of an internal fault. The main drawback of this technique is, of course, that it does not test feature interactions. Reusable component testing is actively used in industry[9]. It is a simple technique that scales well, and that can easily be applied today without any major training or special software. As it exercises commonalities in a product line it will find some of the errors early. The authors noted that several companies reported having partly used reusable component testing in experience

8 Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
9 Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. A Survey of Empirics of Strategies for Software Product Line Testing. In Lisa O'Conner, editor, *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, ICSTW '11*, pages 266–269, Washington, DC, USA, 2011. IEEE Computer Society.

reports: Dialect Solutions reported having used it to test a product line of Internet payment gateway infrastructure products[10]. Testo AG reported having used it to test their product line of portable measurement devices for industry and emission business[11]. Philips Medical Systems reported having used it to test a product line of imaging equipment used to support medical diagnosis and intervention[12]. Ganesan et al.[13] is a report on the test practices at NASA for testing their Core Flight Software System (CFS) product line. They report that the chief testing done on this system is reusable component testing.

Feature Interaction

As just mentioned, testing one feature in isolation, as is the case for reusable component testing, does not test whether the feature interacts correctly with other features. Two features are said to interact if their influence each other in some way[14]. Kästner et al.[15] addressed the optional feature problem related to feature interactions. They mention an example where two features are mutually optional, but where one is implemented differently depending on whether the other is included. One example they consider is a database system with the feature ACID and the feature STATISTIC. The latter feature will include the statistic "committed transactions per second" that only makes sense if the feature ACID is included because it facilitates transactions. They note that feature interactions can cause unexpected behavior for certain combinations of features. An often mentioned example in telecommunications is the two features call waiting and call forwarding. When both are active, which is to take an incoming call? Batory et al.[16] proposed that whenever two features f and g are included in a product, not only are they composed, but so is their interaction How frequent are feature interactions? n features may be combined in O(2n) ways; however, according to Liu et al. experience suggests that interactions among features are sparse. That is, according to experience, most features do not interact. The experience is from, among other sources, the telecommunication system industry. Kästner et al. agrees with Liu et al. in that there usually are more feature interactions than features.

There are techniques proposed in literature that may be able to test feature interactions effectively. Three of these are model-based techniques, reuse-based techniques and subsetheuristics. They are discussed in the following subsections. They are combined with each other and with other techniques in attempts to efficiently test product lines.

Model-based SPL Testing

For model-based product line testing, many different models have been proposed used. Bertolino and Gnesi[17], with their PLUTO method for product line testing, propose modeling the use cases of the product

10 Mark Staples and Derrick Hill. Experiences adopting software product line development without a product line architecture. In Proceedings of the 11th Asia-Pacific Software Engineering Conference, APSEC '04, pages 176–183, Washington, DC, USA, 2004. IEEE Computer Society

11 Dharmalingam Ganesan, Jens Knodel, Ronny Kolb, Uwe Haury, and Gerald Meier. Comparing costs and benefits of different test strategies for a software product line: A study from testo ag. In *Proceedings of the 11th International Software Product Line Conference*, pages 74–83, Washington, DC, USA, 2007. IEEE Computer Society.

12 Gerard Schouten. Philips medical systems. In *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*, pages 233–248. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

13 Dharmalingam Ganesan, Mikael Lindvall, David McComas, Maureen Bartholomew, Steve Slegel, Barbara Medina, Rene Krikhaar, and Chris Verhoef. An analysis of unit tests of a flight software product line. *Science of Computer Programming*, 2012.

14 Jia Liu, Don Batory, and Christian Lengauer. Feature oriented refactoring of legacy applications. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 112–121, New York, NY, USA, 2006. ACM.

15 Christian Kästner, Sven Apel, Syed Saif ur Rahman, Marko Rosenmüller, Don Batory, and Gunter Saake. On the impact of the optional feature problem: analysis and case studies. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 181–190, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

16 Don Batory, Peter Höfner, and Jongwook Kim. Feature interactions, products, and composition. *SIGPLAN Not.*, 47(3):13–22, October 2011.

17 Antonia Bertolino and Stefania Gnesi. Use case-based testing of product lines. *SIGSOFT Softw. Eng. Notes*, 28(5):355–358, 2003.

line with extended type of Cockburn's use cases[18], a textual format for writing use cases. They suggest extending them with variability information. They call this new type of use case Product Line Use Case, or PLUC. Several methods propose modeling product line behavior using UML activity diagrams annotated with variability information. Olimpiew[19] used them as a part of the CADeT method; Reuys et al.[20] used them as a part of the ScenTED method together with UML interaction diagrams and UML use case diagrams, variability was modeled with the Orthogonal Variability Model (OVM)[21]; and Hartmann et al.[22] proposed using UML activity diagrams as a part of their method. They annotate parts of the activity diagrams with product names, and not with features. Thus, their method is limited to situations where all the products are known up front.

State machines have been used for a long time in software engineering. Several propose using them for modeling product line behavior. Cichos et al.[23] proposed creating a single state machine containing all behaviors of all products, called a 150% model. When instantiated according to a configuration, the 150% model yields a 100% model, the behaviour of one product.

Lochau et al.[24] also proposed modeling the behaviour as a state machine, but instead of modeling the union of all product behaviors, they suggested capturing the difference between pairs of state machines using delta modeling. Oster et al.[25] used state machine models to build a 150% model as a part of their MoSo-PoLiTe method.

Reuse

Regression testing techniques can be adapted for product lines. Because any two products *pa*, *pb* of a product line are similar, *pb* can be seen as a development of pa. If we then test pa, we can optimize the testing of *pb* using regression testing techniques. This is product line testing because when delivering product *pn*, the testing of *p1* to *pn*−1 will minimize the effort needed to testing *pn*.

Engström et al.[26] surveyed the use of regression testing technique for product line testing. Uzuncaova et al.[27] proposed using regression testing techniques to minimize test generation time for the next product.

18 Alistair Cockburn. Writing Effective Use Cases. Addison-Wesley Professional, January 2000.
19 Erika Mir Olimpiew. *Model-based testing for software product lines*. PhD thesis, George Mason University, Fairfax, VA, USA, 2008. AAI3310145
20 Andreas Reuys, Sacha Reis, Erik Kamsties, and Klaus Pohl. Derivation of domain test scenarios from activity diagrams. In Klaus Schmid and Birgit Geppert, editors, *Proceedings of the PLEES'03 International Workshop on Product Line Engineering: The Early Steps: Planning, Modeling, and Managing*, Erfurt, 2003.
21 Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
22 Jean Hartmann, Marlon Vieira, and Axel Ruder. A UML-based approach for validating product lines. In Birgit Geppert, Charles Krueger, and Jenny Li, editors, *Proceedings of the International Workshop on Software Product Line Testing (SPLiT 2004)*, pages 58–65, Boston, MA, August 2004.
23 Harald Cichos, Sebastian Oster, Malte Lochau, and Andy Schürr. Model-based coverage driven test suite generation for software product lines. In *Proceedings of the 14th international conference on Model driven engineering languages and systems*, MODELS'11, pages 425–439, Berlin, Heidelberg, 2011. Springer-Verlag.
24 S. Lity, M. Lochau, I. Schaefer, and U. Goltz. Delta-oriented model-based spl regression testing. In *Product Line Approaches in Software Engineering (PLEASE), 2012 3rd International Workshop on*, pages 53–56, 2012.
25 Sebastian Oster, Ivan Zorcic, Florian Markert, and Malte Lochau. Moso-polite: tool support for pairwise and model-based software product line testing. In Patrick Heymans, Krzysztof Czarnecki, and UlrichW. Eisenecker, editors, *VaMoS*, ACMInternational Conference Proceedings Series, pages 79–82. ACM, 2011
26 Emelie Engström, Per Runeson, and Mats Skoglund. A systematic review on regression test selection techniques. *Inf. Softw. Technol.*, 52(1):14–30, January 2010.
27 E. Uzuncaova, S. Khurshid, and D. Batory. Incremental test generation for software product lines. *Software Engineering, IEEE Transactions on*, 36(3):309 –322, may-june 2010.

Svendsen et al.[28] proposed using regression testing techniques to determine whether a test needs to be re-executed for the next product.

Lochau et al.[29] proposed using regression testing techniques to minimize the number of test cases needed to fulfill a model-based coverage criterion for the next product. Dukaczewski et al.[30] proposed using regression testing techniques on annotated textual requirements, as they are more common than test models.

Subset-Heuristics

A product line has surely been thoroughly tested if all products are executed for all possible inputs. The idea of subset-heuristics is to select a subset of the products or of the inputs such that the testing remains (almost) as good as testing all products with all inputs. Some of the challenges of subset heuristics are 1) being able to generate a small subset, 2) being able to generate the subset within a reasonable time and 3) knowing how good a subset selection technique is. Combinatorial Interaction Testing (CIT) can be used to both generate a subset of products and generate a subset of inputs for testing. The technique has been known for a long time, and is discussed and developed extensively. Cohen et al.[31] is one of the earlier papers that introduced an algorithm for generating subsets. The related work will be discussed in depth later. CIT handles constraints between features in a product line and between input variables for system testing.

Kuhn et al.[32] investigated the bug reports for several large systems. They found that most bugs can be reproduced by setting a combination of a few parameters (and no more than six). Thus, they indicate that most bugs can be detected by exercising any combination of a few parameters.

Generating a subset that covers all simple combinations is classified as NP-hard. Algorithms finding non-optimal subsets still keep advancing. Some of these algorithms are discussed later. A contribution of this thesis is an argument for why the subset selection of products of a product line is quick in practice. Another contribution is an algorithm that is currently the fastest algorithm for selecting a subset of products for testing. Both of these contributions relate to product subset selection for testing, and only indirectly to program input subset selection. There are subset heuristic techniques that are not based on CIT. Kim et al.[33] use static analysis to determine a reduced set of products that are relevant for a test. This reduction lessens the combinations of products that need to be tested given a certain test. Kolb[34] proposed using risk analysis to identify a subset of products to test. Scheidemann[35] proposed a technique that selects products such that requirements are covered.

28 Andreas Svendsen, Øystein Haugen, and Birger Møller-Pedersen. Analyzing variability: capturing semantic ripple effects. In *Proceedings of the 7th European conference on Modelling foundations and applications*, ECMFA'11, pages 253–269, Berlin, Heidelberg, 2011. Springer-Verlag

29 Malte Lochau, Ina Schaefer, Jochen Kamischke, and Sascha Lity. Incremental model based testing of delta-oriented software product lines. In Achim D. Brucker and Jacques Julliand, editors, *TAP*, volume 7305 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2012.

30 Michael Dukaczewski, Ina Schaefer, Remo Lachmann, and Malte Lochau. Requirements-based delta-oriented spl testing. In *Product Line Approaches in Software Engineering (PLEASE), 2013 4th International Workshop on*, 2013.

31 D. M. Cohen, S. R. Dalal, A. Kajla, and G. C. Patton. The automatic efficient tests generator. In *Fifth IEEE International Symposium on Software Reliability Engineering, Proceedings of*, page 303–309, 1994.

32 D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo. Software fault interactions and implications for software testing. IEEE Transactions on Software Engineering, 30(6):418–421, 2004.

33 Chang Hwan Peter Kim, Don S. Batory, and Sarfraz Khurshid. Reducing combinatorics in testing product lines. In *Proceedings of the tenth international conference on Aspectoriented software development*, AOSD '11, pages 57–68, New York, NY, USA, 2011. ACM.

34 Ronny Kolb. A risk-driven approach for efficiently testing software product lines. In *Net.ObjectDays 2003. Workshops. Industriebeiträge - Tagungsband : Offizielle Nachfolge-Veranstaltung der JavaDays, STJA, JIT, DJEK*, pages 409—-414, 2003.

35 Kathrin Danielle Scheidemann. *Verifying families of system configurations*. Shaker, 2008.

Oster et al.[36] proposed a method where feature models (FMs) and classification trees (CT) are combined to a Feature Model for Testing (FMT). Classification trees are used in the CT-method[37] for software testing. The FMT model can be used with classification tree tools such as CTE[38] to generate a subset of system tests that also exercise the product line.

7.3. Summary

This section presents two potential tools to improve the state of the practice of configuration test.

The Docker container technology, together with the orchestrations supports on top of it, provide a promising way to reduce the cost of preparing a big number of different configurations, and to improve the automation level of performing testing tasks on top of them.

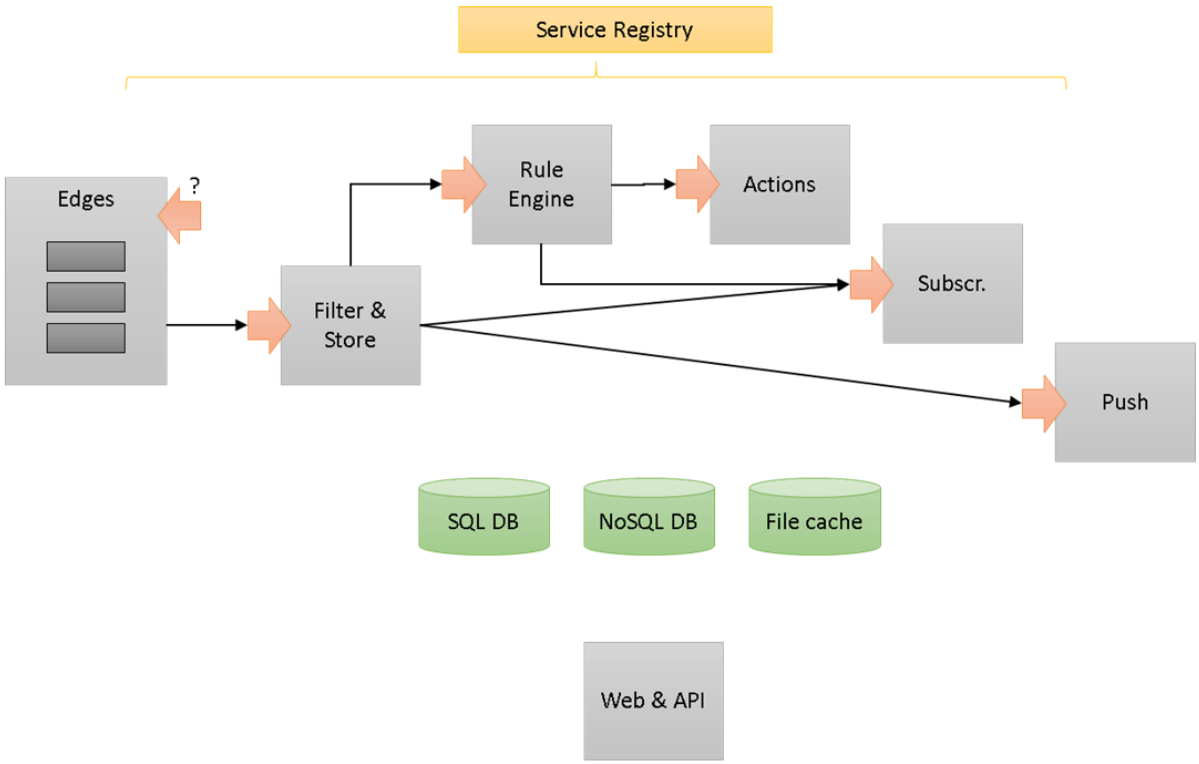
The research approaches around software product line testing provide valuable references on how to use limited configurations under testing to maximize the coverage of the entire configuration space. The potential directions are the efficient management of features, the reuse of testing results and the use of high-level models.

8. Case studies

8.1. Status of configuration testing in TellU

TelluCloud is a cloud services for collection and processing of data, with a focus on IoT. The core of the system went through a refactoring prior to the STAMP project. In this refactoring, it was split up into service components (micro-services), to enable flexible and scalable cloud deployments. The following figure shows the main components of the system.

36 Sebastian Oster, Florian Markert, Andy Schürr, and Werner Müller. Integrated modelling of software product lines with feature models and classification trees. In *Proceedings of the 13th International Software Product Line Conference (SPLC 2009). MAPLE 2009 Workshop Proceedings. Springer, Heidelberg*, 2009.
37 Matthias Grochtmann and Klaus Grimm. Classification trees for partition testing. *Software Testing, Verification and Reliability*, 3(2):63–82, 1993.
38 Sergej Alekseev, Rebecca Tiede, and Peter Tollkühn. Systematic approach for using the classification tree method for testing complex software-systems. In *Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering*, SE'07, pages 261–266, Anaheim, CA, USA, 2007. ACTA Press.



The grey boxes are the microservices. Our focus in STAMP is the branching chain of data processing, starting with receiving data at edges, continuing with filtering, storing and rule engine processing and with various possible outputs, such as sending of messages to users or external systems. Each service component can receive messages from other service components. We support various implementations of the message channels, but the primary intention is to use some form of queue, so that messages are buffered. The service components have access to various forms of storage.

Flexibility and scalability has been the key requirements for this new system architecture, and we need to be able to deploy the system in various configurations and on various infrastructure. We need to support everything from running the whole system on a single machine to running in the infrastructure of the major cloud providers. In a cloud infrastructure, we want to take advantage of the available features, such as auto-scaling and persistent queues. For each of the service component types shown in the figure, there can be multiple instances. A distributed Service Registry keeps track of all instances. Load balancing is needed, and in some cases messages needs to go to a specific instance.

The microservices architecture and large degree of flexibility means there is a large configuration space. We have so far not had any structured approach to configuration testing. We have mainly been testing the specific implementations, and the intended cloud deployment. The micro-service version of the system is not yet in production. We are currently mapping out the configuration space and working on how to efficiently orchestrate deployments.

Service component test plans

We have recently worked out test plans. Relevant for the configuration testing in STAMP are the tests related to the service components, on the component and system levels. Testing on these levels will typically involve sending a set of messages to the component/system, and checking the messages or other types of results produced. Here we have identified the need for the following test types:

- Component testing: This is testing of each service component in isolation. We want two tests for each component type. 1) Functional test, checking that the correct results are produced. 2) Performance test, checking the number of messages processed per second.

- Sharding/load balance testing: This is testing of the scaling and load balance mechanisms, with multiple instances of a service component. A single component type is sufficient for such a test, but we may want to repeat it for different component types. We have identified three types of test. 1) Distribution correctness (that each message goes to the correct instance, in cases where this is significant). 2) Reorganization of shards: Shards are how we split up the messages to a service component type (f.ex. into a number of queues), and service component instances must correctly be reassigned shards each time the number of instances changes. 3) Testing of different configurations (instances, queues, etc.) to find optimal price/performance/load configurations.
- System testing: Here we test the whole system, on a real-world deployment comparable to production. Two types of test: 1) Functional test, checking that messages are correctly processed. 2) Stress/scaling tests.

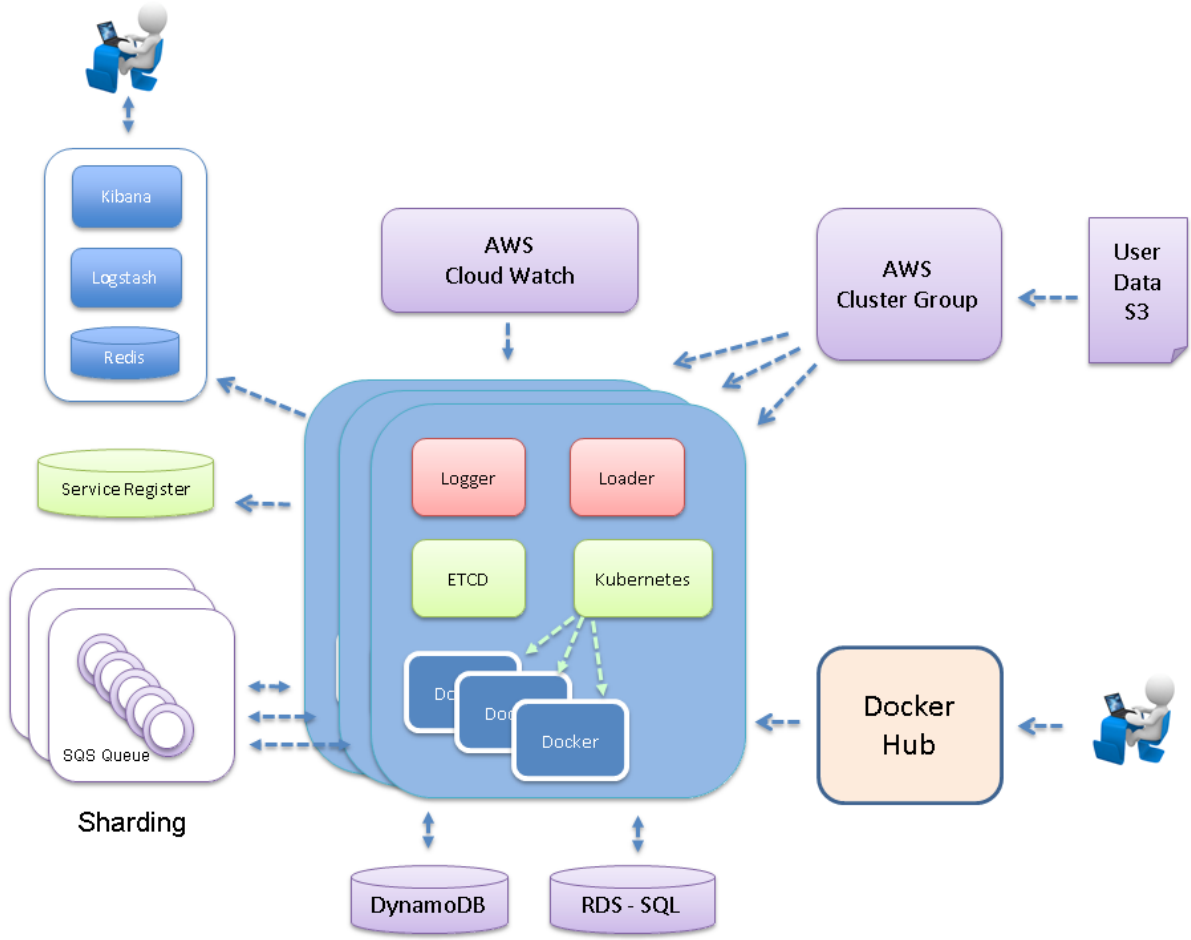
We want component testing to be fully automatic as integration tests run by Maven and Jenkins, using JUnit. This is one level above unit tests, running less often but still regularly (perhaps once per day), checking for bugs and performance problems introduced by code updates. The sharding and load balance mechanisms will typically be quite stable, and full automation of such tests is not a priority. However, we want to make it feasible to run such tests for many different configurations, and to be able to run a test with a single command. The same applies to system testing. Once all other levels of testing are in place, system testing can primarily be done after major changes or before a major release.

We will first implement the system tests. This is our first priority because such tests exercise the total chain including all components, queues and load balance mechanisms. We can then make more focused tests for the specific components and mechanisms, as these will run faster and make it easier to pinpoint the location of the issue when the system fails.

For formalizing and automating the testing as much as possible, we need a way to specify and automate deployment, a way to specify and send messages to the system under test, and a way to capture and check results.

Deployment strategy

Our current production deployment target is Amazon's cloud infrastructure (AWS). We also want to do much of the testing in AWS, but it is also important to be able to deploy and test locally on a developer machine. The following figure shows the current deployment scheme. Here we see the database implementations provided by Amazon (bottom) and its SQS queues we use to implement messaging between service components in this infrastructure (bottom left). And we see the logging stack we use, with Redis, Logstash and Kibana (top left).

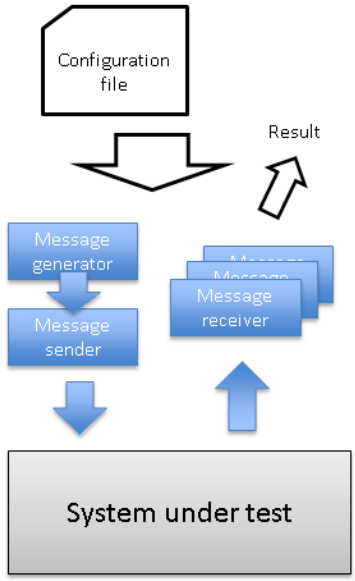


We deploy the service components in Docker containers using Docker Hub. We run these in CoreOS, a light-weight container-focused Linux. CoreOS provides ETCD as a distributed registry, and we have been using this to implement our Service Registry. CoreOS used to come with a cluster management tool called fleet, which was the preferred way to launch containers, and this is what we used initially. However, fleet has been left behind by new developments, and is no longer supported by CoreOS. They now instead advocate the use of Kubernetes. Kubernetes is a more powerful tool for orchestrating cloud deployment, and seems to be the new de facto standard for container orchestration.

Kubernetes appears to be a good match for our case and for configuration testing. It can be used to deploy to different cloud infrastructures, so we maintain great flexibility. It can also be used to deploy locally, with the tool Minikube making it easy to set up a single-node cluster in a VM on any PC. Most importantly, Kubernetes allows specifying deployments in configuration files. This means that much of what used to be external configuration in selecting infrastructure components and resources can be explicitly stated in configuration files. This greatly facilitates configuration testing and automation of deployment.

Our current deployment strategy is to use Kubernetes. We have started using it, initially to do the same deployments as was previously done with fleet. Targets are Minikube and Amazon. As we get a deeper understanding of Kubernetes, we will see if it can help us find better implementations of some of our mechanisms, such as replacing our own sharding and load balancing with Kubernetes mechanisms.

TelluCloud testing tool



As we saw in our test plans, testing of service components entails sending messages in to the system and checking the resulting outputs. We have been developing a flexible component-based testing tool to do this. Our framework specifies three main types of components for messages. A message generator provides the messages to send. A message sender is needed to send them in to the system. A message receiver can receive results. We have made different implementations for these component types, which we can mix and match. The most useful message generator implementation takes a template and possibly a data set as input. The sender and receiver must match the messaging infrastructure deployed. We have senders for HTTP and two queue types (AWS and RabbitMQ). We have receivers for the two queue types, and plan to also make implementations to subscribe to the Subscriber and Push service components.

A test runner coordinates the components, and a separate component keeps track of received messages. It can be used to make assertions programmatically, or write a test report. The testing tool can be configured with files, specifying the components to use and their arguments. The tool can be used in different ways:

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

It can be used to test a queue, a single service component or a whole system. It can be used for stress testing (generating large number of messages), performance testing (tracks messages per second) and functional testing (analyze received messages). An initial version of the testing tool is ready, with the most relevant component implementations. We envisage that development on the tool will continue, and that it could perhaps form the basis of a more general STAMP tool.

One important point is the relation between different tools and configurations involved in running a test. Firstly, we have the configuration of the deployment to test, both in TelluCloud and Kubernetes configuration files. Then the testing tool configuration will specify how/where to send and receive messages, which must match the deployment to test. And then we may have a JUnit test to run the test. Configuration of the testing tool could be specified programmatically from the JUnit code, otherwise it will reside in a file read by the configuration tool. This includes which and how many messages to send, while the JUnit test code must check the aggregated results once ready.

Next steps

Work on deployment with Kubernetes is ongoing. One current task is to map the configuration space, getting a complete overview over all internal and external parameters, how they are specified and which values we want to try for these. Then we will design and implement system tests, and try to automate these as much as possible. This will place us in a position to start testing different configurations. We will need to work on how to coordinate and automate the use of the tools and configurations for testing.

8.2. Status of configuration testing in Atos

In this section we collect the state of practice in Atos Research and Innovation (ARI) department, in the context of ARI participation on R&D&I projects of the H2020 programme, focusing on the techniques and technologies being applied for supporting application deployment and configuration and testing during iterative phases of the development life-cycle, in development and preproduction environments.

Adopted techniques largely depend on different factors, including: i) the background and expertise of the development teams, ii) the kind of application/system being developed, iii) the baseline technologies adopted for development, iv) the deployment and configuration requirements, to cited some reported by development teams.

We structure this survey on the state of practice by grouping adopted technologies by purpose:

Software project management/building tools: Quite popular in ARI is the adoption of project building tools, notably Maven, but recently Gradle being incorporated for automating the building of the software components, which have displaced Ant and script-based technologies. This approach enables full building automation (before deployment) integrated with Continuous Integration (CI). Alternatively, manual building approaches based on IDE facilities (I.e. Eclipse, IntelliJ, Netbeans, Android Studio, etc) are still being adopted, particularly for technologies that impose concrete archiving structure, such as J2EE (WAR, EAR) or Android.

These technologies prepare the deployment units (e.g. archives or assemblies), embedded with application-specific configuration descriptors (e.g. web.xml, *.properties, etc)

Software Configuration Management Tools: different configuration managers are being adopted, including:

- Puppet: is an open-source software configuration management tool for Unix-like and Windows systems that uses a declarative language (or a Ruby DSL) to describe system configurations.
- Chef: is another configuration management tool that uses Ruby DSL for declaring system configuration "recipes". This technology was widely adopted in the FIWARE Ecosystem, being progressively replaced by Docker.
- Ansible: is another opensource configuration management and application deployment automation engine, compatible with Unix-like control machines and both Unix-like and Windows managed nodes. Ansible is used in one ongoing project to configure the inventory of nodes. YAML is used to configure the nodes and their state. YAML can also be used to specify concrete tasks and subtasks on groups of nodes.

These tools keep track and control the changes in the software an apply the configuration needed in order to be successfully deployed in a target environment, not only on the level of software artefacts, but also on the target software environment requirements (I.e. web servers, databases, etc). Configuration tools also assist on the containerization of the final software.

Deployment/Configuration scripting technologies:

Diverse technologies are adopted for supporting manual/automatic deployment and configuration, largely depending on factors described above, particularly on the kind of development platform adopted for development.

For those baseline technologies, such as J2EE, that impose concrete deployment units, they are typically deployed into compatible application containers (I.e. Tomcat, JBoss, Jetty, Glassfish, etc), either manually or automatically. Manually by issuing CLI commands (e.g. SO specific). Examples are scripting languages for Unix, such as Sh, Csh, Bash, etc, or Windows (Batch, Powershell, etc). Scripts (based on these languages)

for automating the configuration and deployment process are also created. They are typically executed within CI/CD deployment projects.

Approaches for configuring applications/systems, before and after deployment, are also technology specific. For J2EE applications, Java property files are adopted for configuration, together with libraries such as Apache Commons Configuration, which can be modify dynamically (e.g. at runtime) and reloaded. Other examples include XML/JSON configuration files, and even language-specific ones (e.g. Clojure).

Other configuration general purpose languages adopted are, such as YAML, a JSON superset, data serialization language, commonly used for configuration files, or the DSL aforementioned.

Continuous Integration (CI) / Continuous Delivery (CD) tools:

- Jenkins/Hudson: Jenkins is an open-source CI/CD platform (forked from Hudson) widely adopted within ARI for software building and testing (CI) and deployment (CD). It is connected through plugins with SCM sources, typically Git (GitLab/GitHub) or SVN, building tools (Maven, Gradle), testing (JUnit, TestNG, etc), QA (Jacoco, Cobertura, SonarQube) and deployment (shell scripts, powershell, Tomcat, Pupplet pipeline, Docker pipeline, Google Play Android publisher etc).
- GitLab: GitLab is extensively used within ARI, not only as a Git hub, but also as a CI/CD tool, supporting building, testing and automated deployment. GitLab CI pipelines are used to test (also integrated with SonarQube QA) and build our software upon pushed code or at schedule). CD phase, after successful CI phase deploy the software. YAML is used to configure GitLab pipelines.
- Travis-CI: is a CI service used to build and test software projects hosted at GitHub. Open-source projects can be tested via travis-ci.org at no charge. Travis-CI is configured using YAML configuration files. Deployment have been supported in Travis-CI within ARI using Bash scripts.

Container Engines adopted for CD:

Container engines adopted in ARI, listed below, act as a container manager and orchestration system, scheduling the creation of the container as well as their deployment.

- Docker: is an open-source framework that automates the deployment of applications inside software containers. In some ARI projects, Docker images are run under Kubernates or Openshit. Docker-compose is used to deploy multiple containers and configure them through environment variables.
- Vagrant: is an open-source software product for building and maintaining portable virtual development environments, which manages all the necessary configurations for the developers in order to avoid the unnecessary maintenance and setup time. Vagrant is used in ARI for testing deployment of packages in VM with different configurations.
- Kubernates: is an open-source service for automating deployment, scaling and management of containerized applications, such as those embedded in Docker containers.
- Openshift: is a supported (by RedHat) distributions of Kubernates, using Docker containers.

Deployment specific technologies:

In the specific niche of deploying on Cloud/Edge environments, different technologies are being adopted, including:

- Tosca: an standard language to describe a topology of cloud based web services, their components, relationships, and the processes that manage them. A TOSCA Simple Profile in YAML is available.
- OpenStack Heat: an orchestration framework to manage the entire lifecycle of infrastructure and applications within OpenStack clouds
- Amazon Cloud Formation: support the creation and management of collection of related AWS resources

- Open Virtual Format (OVF): open standard for packaging and distributing virtual appliances or, more generally, software to be run in virtual machines (managed by hypervisors). Using OVF we can configure different virtual configurations for experimentation.

In the niche of IoT, Resin.io service is being considered to deploy, update and maintain containers on remote devices (e.g. the EDGE).

8.3. Summary

This section describes in detail the state of practice of configuration test in two selected partners, TellU and Atos. Both use cases described in this section are SaaS systems, and involve the architectural and external configurations, with both functional and performance impact on the system. The two use cases all employ automatic tools for the integration and deployment, and have the plan of introducing Docker to future automate the process. These tools are currently use for the development purpose, but we see the potential of using these automatic tools to reduce the cost of preparing configurations for testing purpose. We will use the two use cases as the initial target to start of our experiments on improving and amplifying the configuration test.

9. Future plans for configuration test amplification

Based on the state of practice of configuration test, we will refine the plan in Work Package 2 on investigating the improvement and amplification of configuration test.

In summary, our understanding on the state of the practice of configuration test, within STAMP partners is as follows. The meanings of configuration and configuration test have very wide scopes. People do test multiple configurations, but not in a big number, and the test process is quite immature now, and often involve manual tasks. In such situation, expectation on automatic generation of testing configurations is low.

We identify the following two challenges as the main barrier towards the amplification of configuration test.

- The cost to set-up, maintain and reservation of configurations
- The time to run testing tasks on multiple configurations

Without properly addressing these two challenges, it is difficult to achieve the efficient amplification of configuration test, and even more difficult for development teams to accept the amplification approaches.

Based on the status, we plan to start the investigation of configuration test amplification in a bottom-up way, from a small number of concrete experiments in a narrow scope. In the same time, we will first focus on the efficient execution of configuration tests, before we start investigating the automatic generation of configurations.

We will start the experiments on the use cases provided by TellU and Atos, and focus on the external and architectural configurations that are implicitly defined by the environments. Both partners will provide use cases in the form of SaaS, which are a flexible composition of multiple services. Moreover, even though the systems themselves are not based on microservices style at the moment, the two partners have the plan to redesign the system in the microservices way. The two systems support both the functional reconnection between internal services, and the scaling of some services. In the same time, both systems are dependent to external environments, such as resources, software platforms and third-party services. These features mean that it is import to test sufficient configurations in terms of architectures and environments in order to guarantee both the functional correctness and the performance.

As the first step of the experiments, SINTEF and INRIA will work closely together with ATOS and TellU to set up an automatic environment for configuration test. The environment will comprise a configuration pool with relevant configurations for testing, and support automatic process to run predefined processes on selected configurations from the pool. We will exploit the Docker tools to help construct the configuration pool: Instead of maintaining configurations physically identical to the product instances, we will use Docker to simulate the resources for all the services in the system, and use the Docker-based cluster to simulate the distribution of these services and the connection between them. We will also experiment the usage of image and building scripts provided by Docker in order to balance the cost of maintaining ready-to-test configurations and the time to build the configuration for each testing tasks. On top of the configuration pool, we will identify the

systematic process of running testing tasks on multiple configurations and investigate how to automate the process.

In the beginning, the configuration test environment will be specific to TellU and Atos. The second step will be the investigation on how to generalise the environment into other partners.

The configuration test environment will be the baseline for our further research of automatic generation of configurations in order to amplify the configuration test.

10. Conclusion

This document summarizes the state of the practice on configuration test within the STAMP consortium. The survey reveals that the STAMP partners current do not perform configuration test in a mature way comparing to unit test. The challenges are the cost to prepare and maintain multiple configurations and the time to run complex test tasks on these configurations. Moreover, in different systems, the concept of configuration may have different meanings, and are set up and tested in different ways. Based on the state of the practice, we plan to undertake the WP2 research in a bottom-up way, starting from experiments on part-specific ways to automate and amplify configuration test, and the next step, we will mainly focus on constructing an automatic configuration test environment, as the baseline for automatic generation of more configurations.

11. Acronyms

EC	European Commission
SPL	Software Product Line
JEE	Java Enterprise Edition
QA	Quality Assurance
SaaS	Software as a Service
WP	Work Package
WPL	Work Package Leader
IDE	Integrated Development Environment