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

Title	WP2 – D2.5 – Final Report on Configuration Testing Amplification
Date	Nov. 29, 2019
Writer	Mohamed Boussaa, Activeeon Franck Chauvel, SINTEF Enrique Garcia-Ceja, SINTEF Brice Morin, SINTEF
Reviewers	Benoit Baudry, KTH Lars Thomas Boye, TellU Xavier Devroey, TUDelft Caroline Landry, INRIA
Online version	https://github.com/STAMP-project/docs-forum/blob/master/docs/d25_final_report_configuration_testing.pdf

Table Of Content

1 Executive Summary

This document accompanies the CAMP tool developed in the H2020 STAMP project. CAMP, the main delivery of the Work Package 2, enables configuration test amplification: Provided a template test and a variability model, CAMP generates alternative environments for the system under test. We gather hereafter its user manual, its developer manual and we report about how CAMP can help to find undocumented issues in two open-source projects, namely the Atom text editor and the Sphinx documentation generator.

2 Revision History

Date	Version	Author	Comments
Nov. 4, 2019	0.1	Franck Chauvel (SINTEF)	Outline of the deliverable
Nov. 5, 2019	0.2	Franck Chauvel, Enrique Garcia-Ceja (SINTEF)	Fleshing out the main sections
Nov 6, 2019	0.3	Brice Morin (SINTEF), Mohamed Boussaa (Activeon)	Wrote section about instrumentation to capture and analyze traces.
Nov. 15, 2019	0.4	Franck Chauvel (SINTEF)	Account for reviewers' comments
Nov. 19, 2019	0.5	Franck Chauvel (SINTEF)	Account for comments from the second round of reviews
Nov.28, 2019	0.6	Franck Chauvel (SINTEF)	Account for comments from the second round of reviews
Nov.29, 2019	1.0	Caroline Landry (INRIA)	Quality check before submission

3 Introduction

This report (Deliverable 2.5) is the final output of Work Package 2 (WP 2) about configuration test amplification. It accompanies the CAMP tool, which enables configuration tests amplification in practice. This report complements and finalizes the previous Deliverable of WP 2. In D 2.1 [1], we surveyed the state-of-the-Practice in configuration testing and investigated which features should CAMP have. In D 2.2 [2], D 2.3 [3] and D 2.4 [4], we incrementally built, evaluated and revised CAMP, accounting for the feedback of our case-study partners. This deliverable D2.5 is intended as the main entry-point for:

- Developers who want to use CAMP to test their systems in multiple environments,
- Developers who want to integrate CAMP as part of another tool and/or extend it with new features.

Configuration tests assess the behavior of software components in a specific environment. CAMP adheres to a broad definition of "environment" that includes third-party components such as databases, message queues, or remote services, but also local programming frameworks (e.g., software libraries), as well as software stacks such as language interpreters, operating systems, etc. As software systems become more and more distributed, the complexity of their execution environment grows exponentially. Manually building all possible configurations is no longer possible.

Configuration test amplification aims at improving the effectiveness of configuration testing by reducing the effort needed to come up with alternative environments. To this end, the STAMP project developed the CAMP tool, which let the user define a "template" environment together with a variability model that details changes of interest. With these two inputs, CAMP can generate all possible variants of the template environment and gather evidences that the SUT consistently behave as expected.

We focus hereafter on the CAMP tool itself, and we detail how to install and use the tool, but also how to contribute to the code base. CAMP is an open-source Python 3 application. CAMP relies on the Z3 constraint solver (by Microsoft, released under MIT License) to compute all possible configurations, as well as on Docker to deploy and test each configuration in an isolated environment. CAMP represents around 7 000 lines of code (LoC), hosted on Github[1], written by more than 10 contributors and has been released 30 times. At the time of writing, the latest version is CAMP v0.9.0, released on Nov. 19, 2019 under the MIT License.

We structured this report as follows. Section 6 explains how to install CAMP using Docker, using a dedicated script, or manually. Section 7 explains how to use CAMP to amplify configuration testing, using a simple Java web service as running example. Section 8 details the internal architecture of the CAMP tool and provides documentation for external contributors. Finally, Section 9 shows how amplification of configuration testing helps to reveal unknown issues in two existing open-source projects, namely the Atom text editor and Sphinx documentation generator.

1 See https://github.com/STAMP-project/camp

4 References

[1] H. Song, B. Morin, L. T. Boye, and J. Gorroñogoitia Cruz, "Report on the State of Practices for Configuration Testing," STAMP Consortium D2.1, 2017.

[2] H. Song, J. Gorroñogoitia Cruz, V. Massol, M. Audren, and A. Vasilevskiy, "Initial prototype on configuration test amplification," STAMP Consortium D2.2, 2017.

[3] A. Vasilevskiy, H. Song, V. Massol, L. T. Boye, and C. Landry, "Enhanced prototype of the configuration amplification and report on the performance," STAMP Consortium D2.3, 2018.

[4] F. Chauvel, B. Morin, V. Massol, D. Gagliardi, and S. Morka, "Consolidated tool for the configuration amplification, selection and execution," STAMP Project D2.4, 2019.

[5] R. C. Martin, *Clean Architecture*: Prentice Hall, 2017.

[6] E. Evans, *Domain-driven design: tackling complexity in the heart of software*: Addison-Wesley Professional, 2004.

[7] F. Chauvel, B. Morin, and E. Garcia-Ceja, "Amplifying Integration Tests with CAMP," presented at the International Synopsium on Software Reliability Engineering (ISREE 2019), Berlin, 2019.

[8] C. Atkinson and T. Kühne, "Rearchitecting the UML infrastructure," *ACM Trans. Model. Comput. Simul.,* vol. 12, pp. 290-321, 2002.

5 Acronyms

API	Application Programming Interface
CI	Continuous Integration
EC	European Commission
FTP	File Transfer Protocol
IDE	Integrated Development Environment
JDBC	Java Data Base Connectivity
JDK	Java Development Kit
JMS	Java Messaging Service
JRE	Java Runtime Environment
JSON	Java Script Object Notation
JTL	JMeter Text Logs
HTML	Hyper Text Markup Language
HTTP	HyperText transfer protocol
LoC	Lines of code
MOF	Meta Object Facilities
OCL	Object Constraint Language
OS	Operating System
SUT	System under test
UI	User Interface
VCS	Version Control System
YAML	Yet Another Markup Language
XML	eXtensible Markup Language

6 Installation Guide

CAMP is a Python 3 application that uses the Z3 theorem prover and the Docker engine. CAMP itself is packaged as a Docker image but there are three ways to install CAMP:

1. Using Docker (recommended);
2. Using our install script;
3. Manually.

6.1 Using Docker

The fastest solution is to use docker to run CAMP and all its dependencies in one container, using the following command:

```
user@machine$ docker run --name camp \
                         -it \
                         -v /var/run/docker.sock:/var/run/docker.sock \
                         -t fchauvel/camp:latest bash
root@9dd7e1f061ce:/camp# cd samples/java
root@9dd7e1f061ce:/camp# camp generate -d .
```

As shown above, the CAMP image already contains the code source and the examples in the "samples" directory. Note that we share the docker daemon of the host with the CAMP container (see the "-v /var/run/docker.sock:..." option). CAMP can therefore invoke docker and create “sibling” containers.

This command will fetch the CAMP Docker image named fchauvel/camp:latest from Docker Hub and run it with your working directory ($pwd) mounted in the container at /workspace.

We follow the following convention for tags on our Docker images: "vX.Y.Z" defines the version of CAMP that is running, "latest" defines the latest released of CAMP (i.e., the highest version vX.Y.Z), "dev" defines the latest commit that passed the CI checks.

6.2 Using the install script

Another way is to use our install script, which you can fetch and execute as follows:

```
user@machine$ \curl -L https://github.com/STAMP-project/camp/raw/master/install.sh \
  | sudo bash -s -- --install-z3 --z3-python-bindings /usr/lib/python3.5
```

Here, we specified where the Z3 Python bindings must be installed. Note the --camp-version, which let you install a specific version of CAMP. Here, we installed the development version, directly from the master branch. Once it completed, you can check that CAMP is running as follows:

```
user@machine$ camp -help
```

6.3 Installing Manually

To manually install CAMP, we must first install its dependencies, which are Python 3, Z3 and Docker.

Python 3 is installed on many Linux distributions. Should you need to install it yourself, please follow the instructions given on the Python website for your environment.

Similarly, we refer you to the Docker website for instructions about how to install Docker[2].

2 See https://docs.docker.com/

First, download and unzip the last version of the Z3 solver from the GitHub releases. At the time of writing, the last version is Z3 4.7.1. Then, unzip the archive that fits your platform in a directory of your choice, I unzipped it in the following.

```
$ cd /root
$ wget https://github.com/Z3Prover/z3/releases/download/z3-4.7.1/z3-4.7.1-x64-debian-8.10.zip
$ unzip z3-4.7.1-x64-debian-8.10.zip && mv z3-4.7.1-x64-debian-8.10 unzipped
```

Now, create the directory that will contain the Z3 binaries. I install it within the Python3.5 distribution in /usr/lib/python3.5, and copy the relevant executable file, the Z3 libraries, and the Z3 Python bindings.

```
$ mkdir -p /usr/lib/python3.5/z3/lib
$ cp unzipped/bin/z3 /usr/lib/python3.5/z3/lib/
$ cp unzipped/bin/lib* /usr/lib/python3.5/z3/lib/
$ cp -rf unzipped/bin/python/z3 /usr/lib/python3.5/
$ ln -s /usr/lib/python3.5/z3/lib/z3 /usr/bin/z3
$ rm -rf unzipped
```

You can now check that Z3 is available, by checking its version number as follows:

```
$ z3 --version
Z3 version 4.7.1 - 64 bit
```

You can also check the Z3 Python bindings are properly set up by executing this Python one-liner:

```
$ python -c 'import z3; print(z3.get_version_string())'
4.7.1
```

If you are installing CAMP in order to modify it, you may want to create virtual environments that include the Z3 bindings we just installed globally. To do so use --system-site-packages option when you create your virtual environment, as follows:

```
    $ virtualenv -p /usr/bin/python3.5 --system-site-package .venv3.5
    $ source .venv3.5/bin/activate
    (.venv2.7) $ python -c 'import z3; print(z3.get_version_string())'
    4.7.1
```

Finally, we can install CAMP using any other Python application. As for most of Python applications, the simplest way to install it is using PIP as follows:

```
$ pip install -U https://github.com/STAMP-project/camp.git@master#egg=camp
```

You can check the installation by running:

```
$ camp --version
CAMP v0.7.0 (MIT)
Copyright (C) 2017 -- 2019 SINTEF Digital

 - Linux 4.9.0-8-amd64 (debian 9.8)
 - Python 3.5.3.final.0
 - Z3 4.7.1
 - Docker version 18.09.4, build d14af54
 - docker-compose version 1.23.0, build c8524dc1
```

7 User Manual

We consider below a sample Java webservice called "Greetings" and its tests, as a running example of an application which we would like to amplify. This application runs on top of a typical Java stack, including an application server (Tomcat), running on top of Java runtime environment (JRE). CAMP's amplification follows the three steps shown on below. We show below how to generate all possible configurations, how to realize and, how to execute them.

.

Figure 1 The CAMP amplification process: Generate interesting configuration models, realize deployable configurations, and finally execute each configuration.

7.1 Generating Configurations

The *greetings* application has four main components:

1. The tests component contains a single integration test that invokes the greeting service and checks that the HTTP response status is 200 (OK).
2. The *Greeting* webapp, a typical Java web service running on top of a Servlet container such as Tomcat or Glassfish.
3. The Tomcat application server, which serves our Greeting Services.
4. The JRE, which underlies the execution of any Java programs and the execution of Tomcat for that matter.

We can describe the relationships between these components into a variability model, where we specify the things we would like to vary. This variability model is a YAML file that captures all the possible components and their relationships. Note that these actually represent "types" or "classes" of components, that CAMP will later instantiate to find possible configurations, that are possible assemblies.

```
goals:
  running:
    - IntegrationTests

components:

  tests:
    provides_services: [ IntegrationTests ]
    requires_services: [ Greetings ]
    implementation:
      docker:
        file: tests/Dockerfile

  greetings:
    provides_services: [ Greetings ]
    requires_features: [ ServletContainer ]
    implementation:
      docker:
        file: greetings/Dockerfile

  tomcat:
    provides_features: [ ServletContainer ]
    requires_features: [ JRE ]
    variables:
      version:
        values: [ v7, v8, v9 ]
    implementation:
      docker:
        file: tomcat/Dockerfile

  jre:
    provides_features: [ JRE ]
```

```
    implementation:
      docker:
        file: jre/Dockerfile
```

Our variability models define all the components previously mentioned: tests, greetings, tomcat, and the JRE. The relationships between them are constrained by the services and features they provide and require. In CAMP a service is a capability that can be used by a remote component. By contrast, a feature is a capability that can only be used by other co-located components, that is, components deployed into the same container, or software stack. For instance, the *greeting* component provides a service named "Greetings", which is required by the test component. These two components must therefore belong to two separate containers. Further, the *greetings* component requires a feature named "ServletContainer", which the tomcat component provides. These two components must therefore be deployed into the same container.

Services and features enable a first form of variability as multiple components may provide a given feature or service. Components may also expose variables, that are explicit variation points, which can have multiple values. For instance, the tomcat component has a "version" variable that can have three values, namely "v7", "v8" or "v9" symbolizing versions 7, 8 and 9, respectively.

Given these variation points, CAMP can now generate all the possible configurations. We can use the following command:

```
user@machine$ camp generate -d . --all
CAMP v0.7.0 (MIT)
Copyright (C) 2017 -- 2019 SINTEF Digital

Loaded './camp.yaml'.
  metamodel = load_yaml(data)

 - Config. 1 in './out/config_1/configuration.yml'.
   Includes greetings, jre, tests, tomcat (v7)...

 - Config. 2 in './out/config_2/configuration.yml'.
   Includes greetings, jre, tomcat (v8), tests...

 - Config. 3 in './out/config_3/configuration.yml'.
   Includes greetings, jre, tomcat (v9), tests...

That's all folks!
```

Provided that the graphviz[3] and imagemagick[4] tools are installed, we can visualize the generated configurations as follows. The result is shown by Figure 2 below.

```
user@machine find . -name "*.dot" | xargs -I file dot -Tpng file -o file.png
user@machine find . -name "*.png" | tr '\n' ' ' | montage  -label '%d/%f' @- -geometry 600x600
configurations.png
```

3 See http://graphviz.org/

4 See https://imagemagick.org/index.php

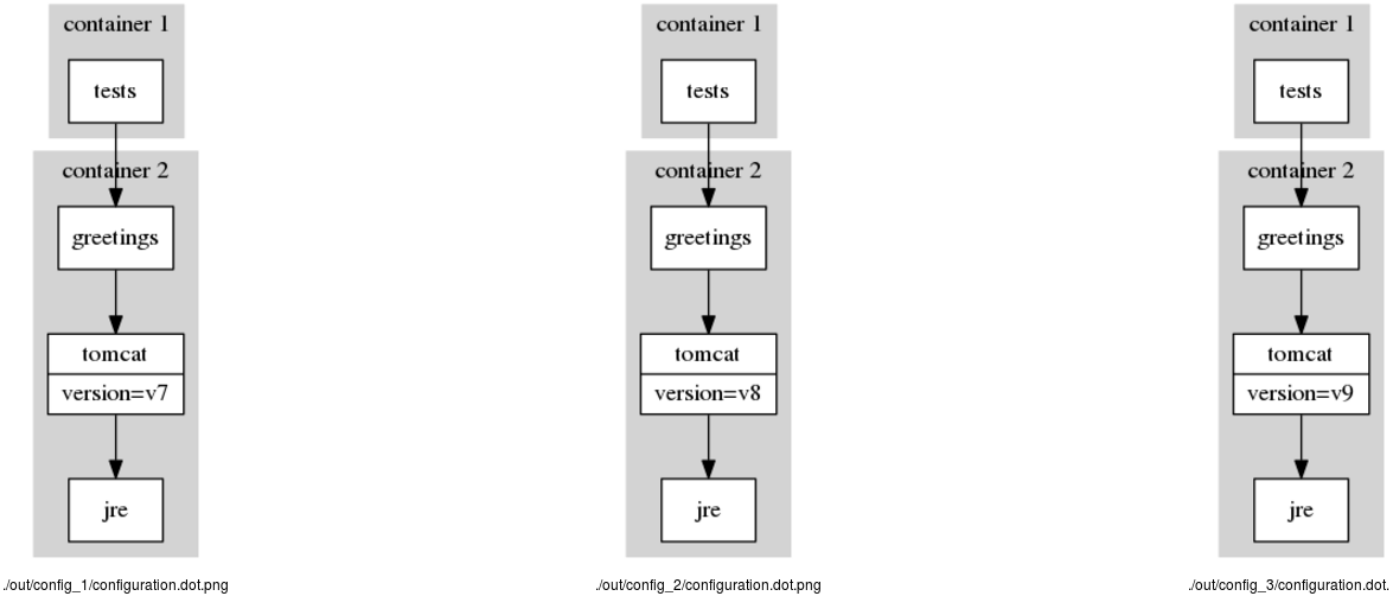

Figure 2 The three configurations generated by the CAMP for the *Greetings* application

7.2 Realization

At this stage, CAMP has only generated models of each configuration. These models are YAML files named "configuration.yml" located in the "out/config_N/" directory. The next logical step is therefore to transform these models into real deployable configurations, a step we call "realization".

To realize configurations, CAMP needs a template for each component described in the variability model, as well as a template orchestration. CAMP uses Docker as a container technology although alternative such as Kubernetes applies in principle. Only the execution phase (see Section 7.4) strictly relies on the Docker and the docker-compose tool. We refer you to D 2.4 [4] for an example of use with Kubernetes. We describe below the structure of each template needed to realize the three configurations of our Greetings service (see Figure 2).

7.2.1 Template Orchestration

We use the docker-compose notation to describe how the services are orchestrated, although the *Greetings* example is simple because it includes only a single service and its test client. Our orchestration, which we describe in the file "template/docker-compose.yml" is organized as follows :

```
version: "3"

services:

  tests:
    build: ./tests
    command: echo Ready!
    depends_on:
      - greetings

  greetings:
    build: ./greetings
    ports:
      - "9080:8080"
```

We can see that our orchestration refers to docker containers to be built from directories named "greetings" and "tests", respectively.

7.2.2 Template for Greetings

Creating the template for the *greeting* service boils down to gathering all the source needed to deploy it on a Docker container. To do so, we need the source code of the service itself, as well as a docker file that details how to compile, package and deploy it into a Tomcat instance. Following the orchestration, we place all this content into the folder "template/greetings". This directory is laid out as follows.

```
user@machine$ tree
.
├── Dockerfile
├── pom.xml
└── src
    └── main
        ├── java
        │   └── org
        │       └── samples
        │           └── GreetingService.java
        └── webapp
            └── WEB-INF
                └── web.xml

7 directories, 4 files
```

The source code of the *Greeting* service is a single Java class (GreetingService.java) placed in the file src/main/java/org/samples. We also define a Maven[5] project descriptor (pom.xml) that automates compilation and packaging.

Our deployment descriptor is a two-stage Docker file. In the first stage, we install all the tools we need to compile and package our Java webapp, that is Maven. Then we copy the source into the container and use maven to compile and package the webapp. In the second stage, we copy the resulting ".war" file from the second container onto the "webapps" folder of our underlying tomcat server. Note the "FROM" statement of this second-stage container: "camp/runtime". It indicates that CAMP will dynamically compute which image shall be used for a particular configuration, because configurations will use different Tomcat versions. We list below the code of our Greetings Dockerfile:

```
FROM openjdk:8-jdk-stretch as builder

RUN apt-get update && \
  apt-get install -y --no-install-recommends  \
  maven=3.3.9-4 && \
     apt-get clean && \
     rm -rf /var/lib/apt/lists/*

RUN mkdir greetings
WORKDIR /greetings

COPY . /greetings
RUN mvn clean package
RUN mv target/greetings-1.0-SNAPSHOT.war target/greetings.war

# Step 2: Deploy the WAR file into a Tomcat instance
FROM camp/runtime

COPY --from=builder /greetings/target/greetings.war /usr/local/tomcat/webapps/greetings.war

CMD ["/usr/local/tomcat/bin/catalina.sh", "run"]
```

7.2.3 Template for the Tests

We proceed in similar fashion for the "tests" component, which is also a Java project whose layout is as follows:

```
$ tree
.
├── Dockerfile
```

5 See https://maven.apache.org/

```
├── pom.xml
└── src
    └── test
        └── java
            └── org
                └── samples
                    └── GreetingServiceTest.java

5 directories, 3 files
```

The Dockerfile specifies how to compile, package and install our test application on a container as follows:

```
FROM openjdk:8-jdk-stretch

RUN apt-get update && \
 apt-get install -y --no-install-recommends  \
 maven=3.3.9-4 \
 && \
   apt-get clean && \
 rm -rf /var/lib/apt/lists/*

WORKDIR /tests
COPY . /tests
```

7.2.4 Template for Tomcat

We proceed in a similar manner for our Tomcat component. The only difference is that we have chosen not to build Tomcat from its sources, but to directly download pre-built binaries from the Apache archive website. Our Tomcat Dockerfile therefore simply downloads the tarball from the Apache archive, decompresses it, and copies the files into the "/usr/local/tomcat" folder as follows. Here as well, we do not know which Docker image our container will be built from, so we use the "camp/runtime" marker in the "FROM" statement to allow CAMP to override it. Note as well how we extracted the major and minor versions numbers of Tomcat into two variables. This will further help us automate changes in this file (see Section 7.2.6).

```
FROM camp/runtime

ARG TOMCAT_MAJOR=7
ARG TOMCAT_REVISION=0.96
ARG TOMCAT_VERSION=${TOMCAT_MAJOR}.${TOMCAT_REVISION}

RUN apt-get -y update \
        && apt-get -y upgrade \
        && apt-get -y install wget

RUN mkdir /usr/local/tomcat \
        && wget http://www-eu.apache.org/dist/tomcat/tomcat-
${TOMCAT_MAJOR}/v${TOMCAT_VERSION}/bin/apache-tomcat-${TOMCAT_VERSION}.tar.gz -O
/tmp/tomcat.tar.gz \
        && cd /tmp && tar xvfz tomcat.tar.gz \
        && cp -Rv /tmp/apache-tomcat-${TOMCAT_VERSION}/* /usr/local/tomcat/
EXPOSE 8080

CMD /usr/local/tomcat/bin/catalina.sh run
```

7.2.5 Template for JRE

Finally, we must also create Dockerfile for the JRE component. We decide to install the OpenJDK directly from the Debian package repositories as follows.

```
FROM debian:stretch

RUN apt-get -y update \
        && apt-get -y upgrade \
        && apt-get -y install openjdk-8-jre
```

Here, we now can decide which Docker image shall be used to build this container, because this container is the bottom of our software stack and there is no way CAMP can change anything underneath (see Figure 2).

7.2.6 Realization Actions

The final step needed to allow CAMP to realize the configurations is to specify how variables and other variations shall be accounted for. To this end, we extend our variability model and add a new realization section for the "version" variable of the Tomcat component, as follows:

```
tomcat:
    provides_features: [ ServletContainer ]
    requires_features: [ JRE ]
    variables:
      version:
        values: [ v7, v8, v9 ]
        realization:
         - targets: [ tomcat/Dockerfile ]
           pattern: "TOMCAT_MAJOR=7"
           replacements:
             - TOMCAT_MAJOR=7
             - TOMCAT_MAJOR=8
             - TOMCAT_MAJOR=9
         - targets: [ tomcat/Dockerfile ]
           pattern: "TOMCAT_REVISION=0.96"
           replacements:
             - TOMCAT_REVISION=0.96
             - TOMCAT_REVISION=5.46
             - TOMCAT_REVISION=0.26
    implementation:
      docker:
        file: tomcat/Dockerfile
```

This new realization entry specifies what has to be done to the template with respect to the value that CAMP chooses for the version variable. We specify here two substitutions, which both target the Dockerfile of our Tomcat template. The first substitution finds and replaces occurrences of the pattern "TOMCAT_MAJOR=7" by a different one reflecting alternative major versions. The second substitution finds and replaces occurrences of the pattern " TOMCAT_REVISION=0.96" by alternative revision numbers. We see here that the work we did on the Tomcat Dockerfile simplify how we enact the changes of versions.

We can now ask CAMP to realize all the configurations as follows:

```
user@machine$ camp realize -d .
CAMP v0.7.0 (MIT)
Copyright (C) 2017 -- 2019 SINTEF Digital

Loaded './camp.yaml'.
Loading configurations from './out' ...
 - Built configuration './out/config_1.
 - Built configuration './out/config_2.
 - Built configuration './out/config_3.

That's all folks!
```

At this stage CAMP has generated three configurations in the out directory. Each configuration includes a modified copy of our template folder. For instance, in the "out/config_3/" folder, we found the following files:

```
user@machine$ tree
.
├── configuration.dot
├── configuration.dot.png
├── configuration.yml
├── docker-compose.yml
└── images
    ├── build_images.sh
    ├── greetings_0
    │   ├── Dockerfile
    │   ├── pom.xml
    │   └── src
    │       └── main
    │           ├── java
    │           │   └── org
    │           │       └── samples
    │           │           └── GreetingService.java
    │           └── webapp
```

```
    │                └── WEB-INF
    │                    └── web.xml
    ├── jre_0
    │   └── Dockerfile
    ├── tests_0
    │   ├── Dockerfile
    │   ├── pom.xml
    │   └── src
    │       └── test
    │           └── java
    │               └── org
    │                   └── samples
    │                       └── GreetingServiceTest.java
    │
    └── tomcat_0
        └── Dockerfile

17 directories, 14 files
```

Looking at the Tomcat Docker file (located in the folder "images/tomcat_0/"), we see it includes the changes we needed. Its initial FROM statement has been updated and now refers to an image named "camp-jre_0" and the values of the variables TOMCAT_MAJOR and TOMCAT_REVISION have been updated to refer to Tomcat version 9.0.27.

```
FROM camp-jre_0

ARG TOMCAT_MAJOR=9
ARG TOMCAT_REVISION=0.27
ARG TOMCAT_VERSION=${TOMCAT_MAJOR}.${TOMCAT_REVISION}

RUN apt-get -y update \
        && apt-get -y upgrade \
        && apt-get -y install wget

RUN mkdir /usr/local/tomcat \
        && wget http://archive.apache.org/dist/tomcat/tomcat-
${TOMCAT_MAJOR}/v${TOMCAT_VERSION}/bin/apache-tomcat-${TOMCAT_VERSION}.tar.gz -O
/tmp/tomcat.tar.gz \
        && cd /tmp && tar xvfz tomcat.tar.gz \
        && cp -Rv /tmp/apache-tomcat-${TOMCAT_VERSION}/* /usr/local/tomcat/
EXPOSE 8080

CMD /usr/local/tomcat/bin/catalina.sh run
```

Similarly, the Dockerfile of the *Greeting* service has been updated as well. The FROM statement of the second stage now points towards an image named "camp-tomcat_0".

```
FROM openjdk:8-jdk-stretch as builder

LABEL stage="intermediate"

RUN apt-get update && \
 apt-get install -y --no-install-recommends  \
 maven=3.3.9-4 && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*

RUN mkdir greetings
WORKDIR /greetings

COPY . /greetings
RUN mvn clean package
RUN mv target/greetings-1.0-SNAPSHOT.war target/greetings.war

# Step 2: Deploy the WAR file into a Tomcat instance
FROM camp-tomcat_0

COPY --from=builder /greetings/target/greetings.war /usr/local/tomcat/webapps/greetings.war

CMD ["/usr/local/tomcat/bin/catalina.sh", "run"]
```

These Docker images (camp-jre_0 and camp-tomcat_0) are built by a script that CAMP has generated as well (see the file "images/build_image.sh"). We see below the part that will build the images when we execute the configuration:

```
build_images () {
    docker build --force-rm --no-cache -t camp-tests_0 ./tests_0
    docker build --force-rm --no-cache -t camp-jre_0 ./jre_0
    docker build --force-rm --no-cache -t camp-tomcat_0 ./tomcat_0
    docker build --force-rm --no-cache -t camp-greetings_0 ./greetings_0
}
```

7.3 Instrumentation to Collect Traces

To assess the impact the different environments generated by CAMP have on a given software component, it is possible to instrument this component. This way, it is possible to observe how the logic inside this component is exercised differently in different environments. Observing the detailed logic execute by a component typically depends on the programming language that is used to implement it. We provide support for two complementary approaches:

- A generic approach that capture system calls
- A Java-specific approach that capture Java stacks using an agent

Both approaches can be visualized with Flamegraphs[6]. Figure 3 shows an aggregation of all the Java traces collected on Proactive, Activeon's use cases. Invocations to methods developed by Activeon are shown in purple. An atomic trace is a vertical slice in this flamegraph i.e, a sequence of method invocations, where the bottom-most method invokes then next bottom-most method, etc. until the top-most method, which then returns to the next top-most, etc. back to the bottom most method.

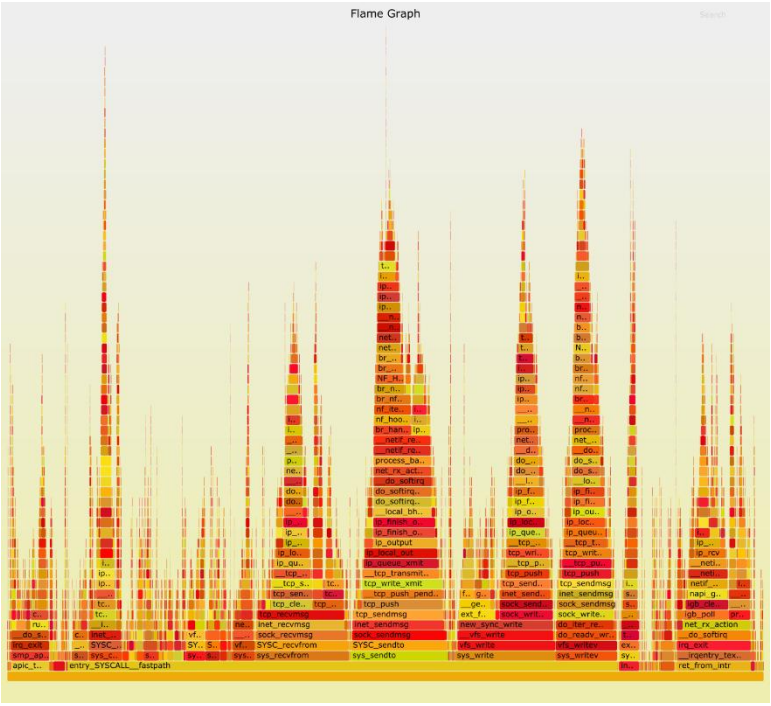

Figure 3 Aggregation of all the Java traces collected on Proactive, Activeon's use cases

Figure 4 shows a diff between one specific configuration of Proactive and the aggregated traces shown previously. Red rectangles indicate differences i.e, methods that have not been invoked by this configuration, but that have been invoked by at least one other configuration.

6 http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html#Java

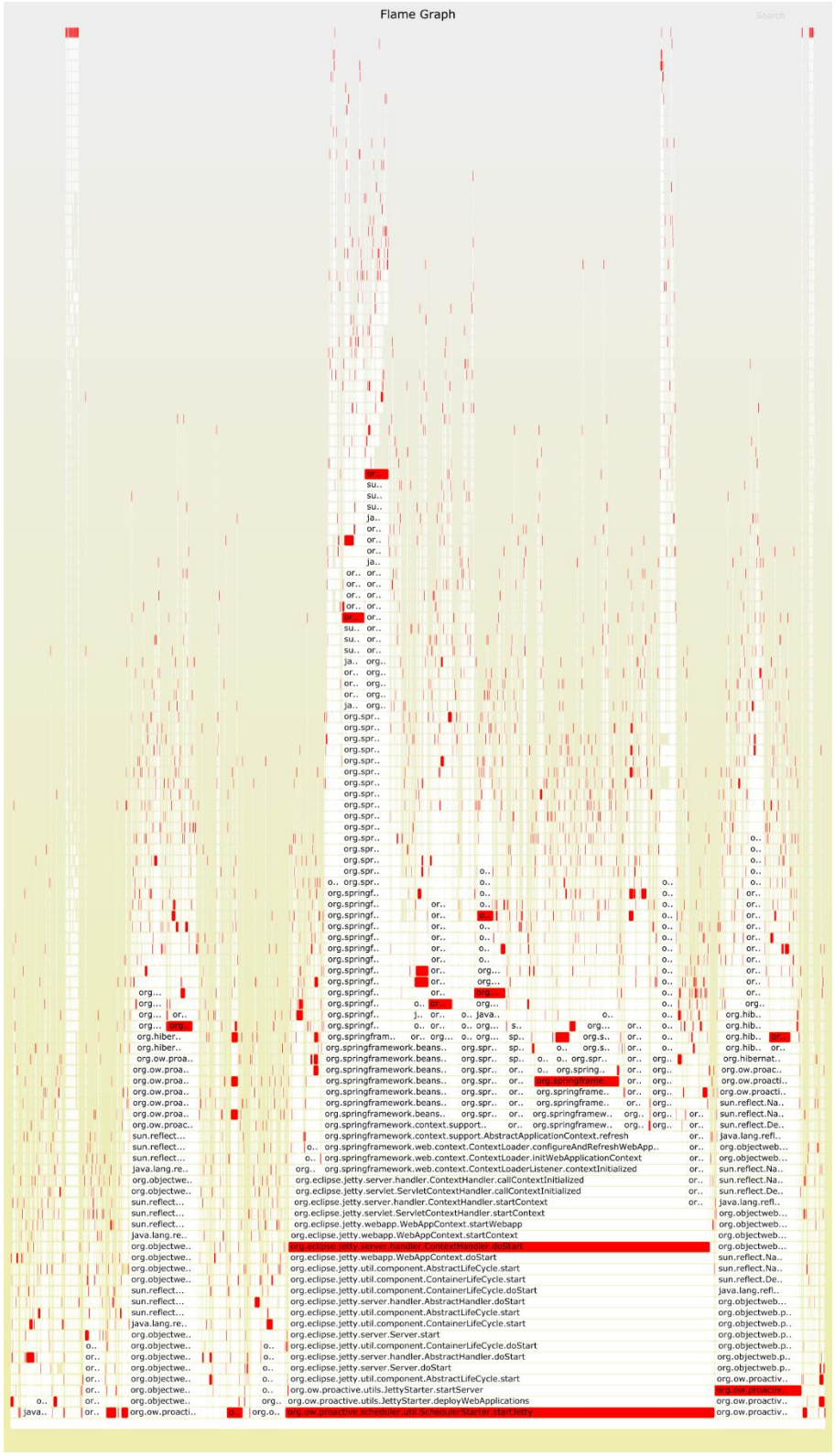

Figure 4 Differences between one specific configuration of Proactive and the aggregated traces shown in Figure 3

Ultimately, all configurations are compared to the aggregated traces, yielding the following output:

```
---- Java traces ----,
Configuration 0 contributed 2190/6255 = 35.00 % of all unique traces
Configuration 1 contributed 2563/6255 = 40.00 % of all unique traces
Configuration 2 contributed 2594/6255 = 41.00 % of all unique traces
---- System traces ----
Configuration 0 contributed 1363/2569 = 53.00 % of all unique traces
Configuration 1 contributed 1238/2569 = 48.00 % of all unique traces
Configuration 2 contributed 1225/2569 = 47.00 % of all unique traces
```

In this example, this shows that no single configuration can fully explain the aggregated traces. In other words, more configurations can trigger more traces.

7.4 Execution

CAMP has now generated configurations that we can deploy, but in order to execute tests and collect the reports, we must again extend our variability models with a little more information. We must specify which component contains the tests, the command used to run these tests, as well as the format and location of the test reports. To this end, we add a "tests" entry to our "tests" component as follows:

```
tests:
    provides_services: [ IntegrationTests ]
    requires_services: [ Greetings ]
    implementation:
      docker:
        file: tests/Dockerfile
    tests:
      command: mvn -B test
      reports:
        format: junit
        location: target/surefire-reports
        pattern: .xml
```

As our test component is also a Java application driven by Maven, the command to run the test is "mvn -B test", which will produce JUnit test reports in the "target/surefire-reports" folder (see Section 7.2.3).

Now we can use the command "camp execute" to execute some or all configurations we have generated. In the following we execute only Configuration 3, as follows:

```
user@machine$ camp execute -d . --include 3
CAMP v0.7.0 (MIT)
Copyright (C) 2017 -- 2019 SINTEF Digital

Loaded './camp.yaml'.
Loading configurations from './out' ...

 - Executing ./out/config_3
   1. Building images ...
      $ bash build_images.sh --build (from './out/config_3/images')
   2. Starting Services ...
      $ docker-compose up -d (from './out/config_3')
   3. Running tests ...
      $ docker-compose run tests mvn -B test (from './out/config_3')
   4. Collecting reports ...
      $ docker ps --all --quiet --filter name=config_3_tests_run_ (from './out/config_3')
      $ docker cp 3fb9ea5ee5a9:/tests/target/surefire-reports ./test-reports (from
'./out/config_3')
      Reading TEST-org.samples.GreetingServiceTest.xml
   5. Stopping Services ...
      $ docker-compose down --volumes --rmi local (from './out/config_3')
      $ bash build_images.sh --cleanup (from './out/config_3/images')

Test SUMMARY:

Configuration                 RUN   PASS   FAIL  ERROR
-----------------------------------------------------
./out/config_3                  1      1      0      0
-----------------------------------------------------
TOTAL                           1      1      0      0
```

```
That's all folks!
```

8 Developer Manual

We detailed below the architecture and the design of CAMP as per v0.8.0 (Nov. 13, 2019). We refer the reader to the online documentation for a comprehensive and updated treatment[7].

CAMP adheres to the clean architecture [5], which separates concerns into different layers. As shown on Figure 5 below, at the core lay entities that represent the configuration testing domain (the domain model in DDD [6]). These entities include for instance, configuration, components, instance, variables, etc. Around are the CAMP use-cases that are: Generate all possible configurations, realize the configurations, and execute these configurations. Around the configurations, are tooling to present the core concepts as well as the input and output of the core services. Finally, the outer layer includes technology-specific facilities to interact with, for instance, the file system or the UI.

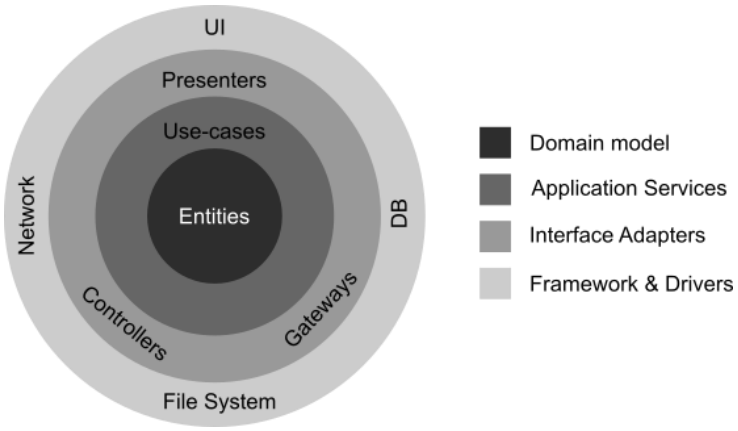

Figure 5 Overview of the clean architecture (inspired from [5])

In CAMP, the external components are the Z3 constraint solver that CAMP uses to compute all the possible configurations; the Docker engine, which CAMP uses to execute each configuration in an isolated environment; and the file system that contains the test template provided by the user.

8.1 Use-cases

Figure 6 shows how the clean architecture translates into the Python classes. The entities are isolated into a separate package and are implemented as plain-old python objects. A controller class intercepts the parameters (provided by the user on the command line) and invokes the appropriate method on the CAMP object, which implements the use-cases as methods and manipulate the entities. The CAMP object delegates to the infrastructure class the instantiation and the persistence of these entities, which, by default, are read and written from and to the disk. The infrastructure has a set of "codecs", which read and write various file formats such as YAML, XML, etc.

7 See https://stamp-project.github.io/camp/

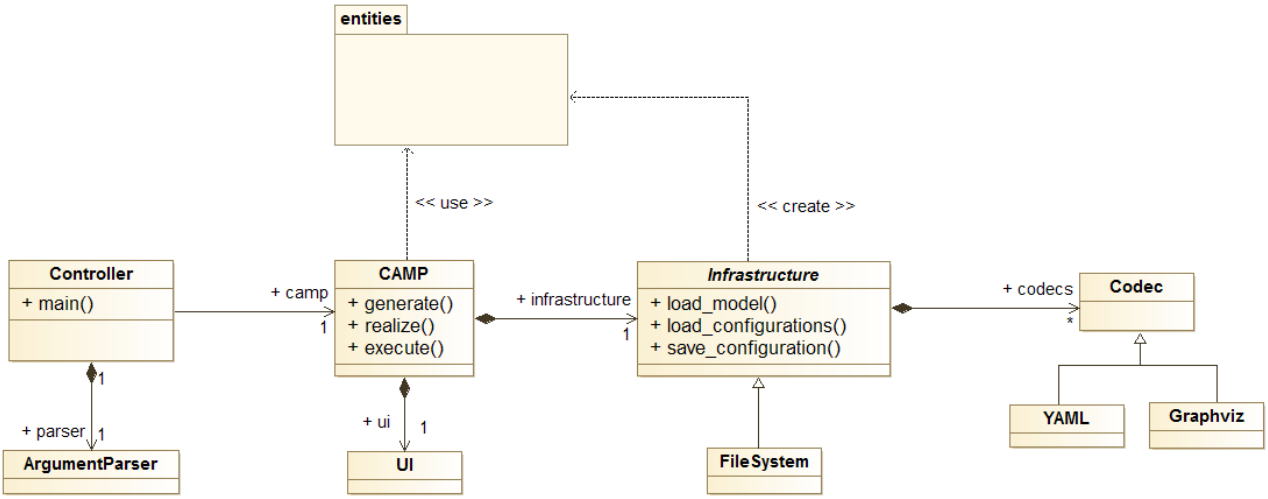

Figure 6 Detailed architecture of the CAMP tool

8.2 Entities

We summarize below the main "concepts" that CAMP uses to carry out configuration testing amplification. We refer the reader to as well as [7] the STAMP Deliverable 2.4 [4] for a comprehensive treatment. CAMP relies on the type/instance pattern [8], whose core idea is to represent, inside the same model, types and their instances. CAMP uses this pattern to let the user define the component types that her application requires and uses a solver to figure out all the possible assemblies of instances that satisfy the domain constraints.

8.3 Development Infrastructure

CAMP is developed according to modern software engineering practices to maintain code quality and minimize the number of defects, including unit and acceptance tests, continuous integration (CI), as well as automated code reviews. The source code is secured on a dedicated git VCS repository (hosted by by Github)[8] , to which, partners of the STAMP project contributed. This repository is connected to the CircleCI continuous integration services and build CAMP so that every commit and pull requests are is built automatically. Figure 7 below illustrates the pipeline that automates the construction and releases of CAMP. We first build a local Docker image that includes CAMP as well as all its dependencies (from the last commit on Github). We then use this image to run multiple unit and acceptance tests. Should all tests pass, the we report code coverage to the Codacy service, and we publish the docker image on DockerHub[9]. CircleCI detect tagged commits and automatically adds a tag on Docker images (e.g, "dev", "latest", "x.y.z"), accordingly.

8 See https://github.com/STAMP-project/camp

9 See https://hub.docker.com/r/fchauvel/camp/

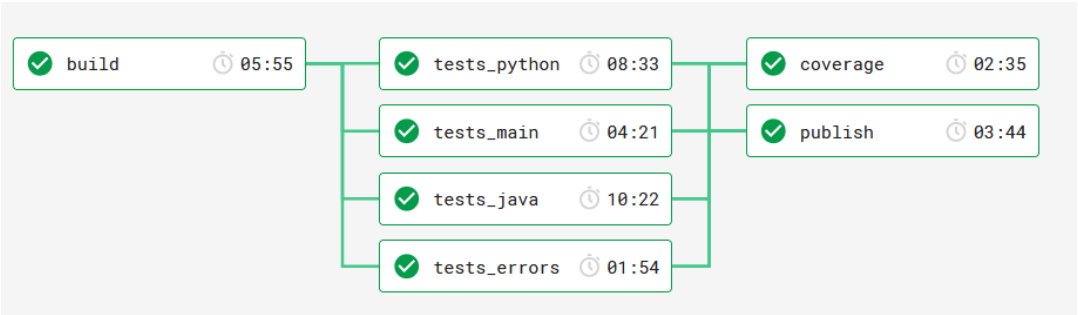

Figure 7 CAMP build pipeline on CircleCI (CAMP v0.8.0)

As per version 0.9.0, CAMP includes about 250 unit tests (see "tests_main" on Figure 7), as well as a dozen acceptance tests ("tests_python", "tests_java" and "tests_errors" on Figure 7). The acceptance tests load complete applications in Python and Java and generate, realize and execute all possible configurations. All these tests cover about 90 % of the CAMP source code. The Codacy service[10] continuously monitors code coverage as well as the overall code quality.

10 See https://app.codacy.com/manual/SINTEF-9012/camp/dashboard

9 Open Source Case-studies

We report below how CAMP helped find hidden dependencies to the environment in existing open-source projects. The two following experiments were initially published in [7]. We applied CAMP on two real systems: Sphinx, mostly implemented in Python, and Atom, mostly implemented in NodeJS. We used CAMP to amplify the direct and non-explicit dependencies of those two projects.

A direct dependency refers to a library, seen as a black-box, which is required to build the source-code. This library may itself have dependencies, which we refer to as indirect dependencies for the source code under investigation, and which we consider out of scope in our experiment.

For a given dependency, say *color* as used in Atom, an explicit version number refers to a specific release, say 1.0.2. An implicit version number resolves to an arbitrary number of explicit versions, and is typically specified as an interval, or a combination of intervals. For example, an implicit version *v* for *color* could be described as >=1.0.1 and <2. This would resolve to versions 1.0.1, 1.0.2 and 1.0.3. Even though the interval is open and theoretically infinite, it will always resolve, at any given time, to a finite number of versions, bound by the latest release.

We selected these two use-cases as two representatives of a large number of open-source projects, which comply to the following requirements:

- Large, popular, and open source. The Sphinx Github repository contains more than 13 000 commits and more than 100 releases, available under a custom open-source license. It has been starred by more than 2 500 users. The Atom Github repository contains more than 36 000 commits and more than 500 releases, available under the MIT license. It has been starred by more than 48 000 users.
- Relies on any number of direct dependencies, any number of them being non-explicit about their version numbers. Sphinx has 16 direct dependencies, 7 of them having implicit numbers. Atom declares more than 154 direct dependencies, 35 of which have implicit version numbers.
- Is, or can easily be, packaged and tested in a Docker container.

Our hypothesis is that for a high-enough number of direct dependencies with implicit versions, it is very likely to find a configuration of the system that is valid according to the specifications but fails the quality assurance (QA) requirements. We claim that CAMP can facilitate the design and operation of experiments aiming at investigating this hypothesis, or similar hypothesis related to integration testing.

9.1 Amplified Configuration Testing for Atom

Atom is "a hackable text editor, built on Electron", mostly implemented in JavaScript (Node.JS) and CSS/HTML. The source code for Atom v1.36.1, released on the 25th of April 2019, relies on 35 dependencies with implicit version numbers.

9.1.1 Experimental Setup

Out of the 35 implicit dependencies needed by Atom and defined in the package.json, we selected 12 dependencies, ignoring all internal dependencies (dependencies to other Atom sub-modules) and ignoring dependencies related to the test infrastructure. In other words, we focused this experiment on the external dependencies provided by 3rd parties and used by Atom's core logic.

Each of those dependencies resolves to at least two versions, and up to 18 versions. We described them in the CAMP variation model (about 170 lines). Altogether, those 12 implicit dependencies yield more than 150 million configurations (i.e., the product of the number of valid versions for the different dependencies). Our hypothesis states that at least one will fail the QA requirements: It will fail more tests than the latest release of Atom (v1.36.1), which failed 0 tests.

Obviously, testing all those 150 million configurations is impractical. We thus run camp generate in coverage mode, so as to drastically reduce the number of configurations, while still ensuring that each and every version for each and every dependency (selected in our experiment) is instantiated in at least one configuration. This yielded 21 configurations only.

9.1.2 Results

Figure 8 exposes a view on the test results for the 21 configurations generated by CAMP. The different axes correspond to the 12 dependencies considered in the experiment. A point on a given axis corresponds to a specific version of that dependency. Versions are in ascending order starting from the center. Points read as follows:

- A grey cross indicates that all configurations where this specific version of that library was involved have failed while executing the tests. We observe that 4 dependencies have such grey crosses for some of their versions. In total, 7 configurations out of the 21 generated by CAMP have raised an error during testing, which resulted in no test report.
- A plain circle indicates that all configurations where this specific version of that library was involved could execute all without raising any error. However, those configurations failed to successfully pass all the tests. For example, none of configurations relying on the three most recent versions of the *marked* dependency, were able to pass all the tests. In total 12 configurations out of the 21 have failed some tests.
- An emphasized circle (a plain circle, doubled by a larger blue circle) indicates that at least one configuration where specific version of that library was involved has passed all the tests. For example, at least one configuration including the latest versions of "yargs", "glob" or "minimatch". In total, only 2 configurations out of the 21 were able to pass all the tests.

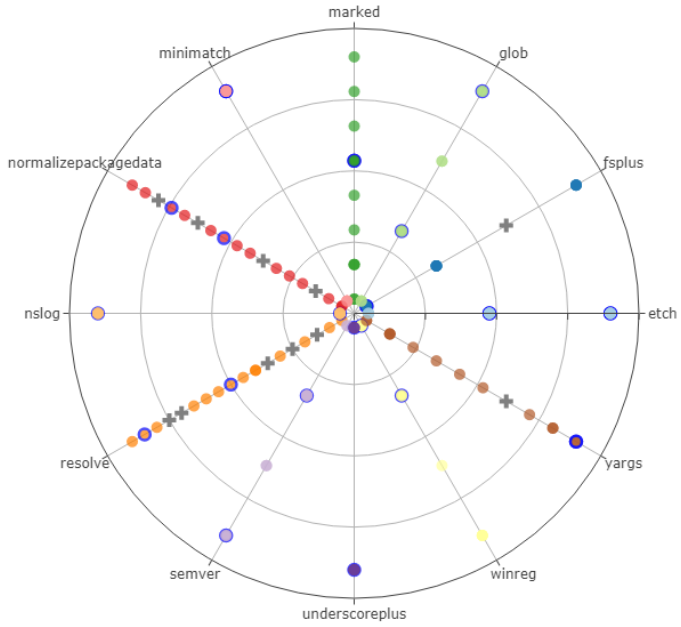

Figure 8 Test results for the 21 Atom configurations. Axes represent libraries and circles specific versions in ascending order starting from the center. A cross indicates that all configurations using that library version failed during testing. Borrowed from [7].

By generating 21 configurations (out of 150 million possible configurations), CAMP was able to verify our hypothesis: 19 configurations out of 21 do not comply to QA requirements, though they all adhere to the developers' specifications.

A detailed discussion about how to improve the quality of those specifications was out of the scope for this analysis, and beyond the scope of CAMP itself. Assuming there is no interaction between those libraries, emphasized circles mean that those specific versions of those specific libraries are safe to use. Other types of points in Figure 8 cannot, and should not, be interpreted too hastily. For example, a grey cross does not necessarily mean that this version of that dependency is responsible for producing an error during the test procedures. Intuitively, one can observe there are 11 of those grey crosses, for only 7 configurations producing an error. However, the fact that only four dependencies have versions with grey crosses implies that those four dependencies can explain all the 7 configuration that yielded errors, which made it impossible

to complete the testing procedure. Those fours dependencies, and how their implicit versions are declared, certainly would need more attention from the developers. A more refined experiment focusing on those dependencies could help developers in figuring out which one (or ones) of those dependencies is/are the actual culprit(s).

9.2 Amplified Configuration Testing for Sphinx

Sphinx is a tool to create documentation for Python projects, mostly implemented in Python. The source code for Sphinx v2.0.1, released on the 8th of April 2019, relies on 7 dependencies with implicit version numbers.

9.2.1 Experimental Setup

Each of the 7 dependencies with implicit version numbers resolves to at least two versions, and up to 20 versions. We described these in a variation model of about 140 lines. Altogether, those 7 implicit dependencies yield 402 300 configurations i.e., the product of the number of valid versions for the different dependencies. Our hypothesis states that at least one will fail the QA requirements: Some will fail more tests than the latest release of Sphinx (v2.0.1), which failed 0 tests.

Again, the total number of possible configurations is impractical for testing each configuration one by one. We thus generated a covering array, yielding 22 configurations, only.

9.2.2 Results

Figure 9 exposes a view on the test results for the 22 configurations generated by CAMP. The different axes correspond to the 7 dependencies considered in the experiment. A point on a given axis corresponds to a specific version of that dependency. Unlike Atom, all the configurations could be tested i.e., a complete test report could be generated for all of them. The size of a circle in this graph corresponds to the percentage of failed tests on all configurations within that dependency and weighted by the number of versions for that dependency. Some versions being more present than others in the 22 configurations, this graph should be interpreted with cautions. We however observe that for a number of dependencies, for example *jinja2*, *alabaster* and *Pygments*, the bigger circles tend to concentrate in the center of the graph, indicating that Sphinx v2.0.1 is no longer compatible with the oldest versions of those libraries, despite what developers still claim in their specifications. We also observe the same pattern for *request*. However, being the dependency with the highest number of versions, it is hazardous to hypothesize that Sphinx is no longer compatible with older versions of *request*. Indeed, each version of request is under-represented compared to other versions of other libraries, yielding very low statistical evidence to support any such claim. For *docutils*, we tend to observe the opposite pattern: the smallest point being associated with the oldest version of that library, while more recent versions are involved in more failed tests. This library having only three versions, each version is well represented in the 22 configurations, so it is reasonable to hypothesize that Sphinx is not forward compatible with respect to *docutils*.

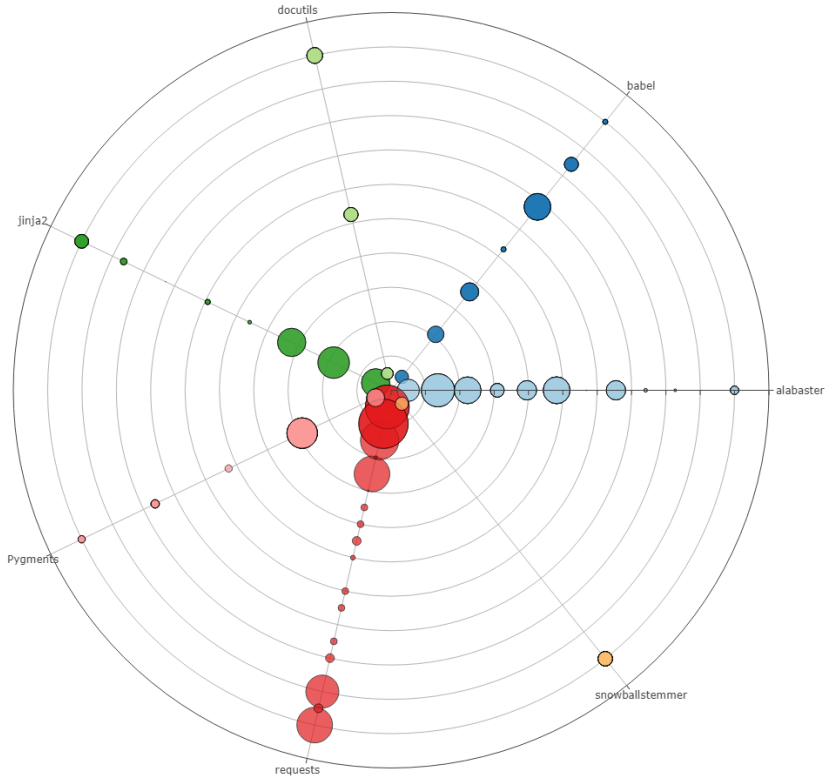

Figure 9 Test results for the 22 Sphinx configurations. Axes represent libraries and circles specific versions in ascending order starting from the center. The size of a circle corresponds to the percentage of failed tests on all configurations within that dependency. Borrowed from [7].

In total, for the 22 configurations:

- Only three configurations were able to pass all tests.
- Six configurations failed between 15 and 20 tests.
- The remaining 13 configurations failed between 1 and 5 tests

This supports our hypothesis. Based on those observations, we conducted a complementary experiment with CAMP where:

- We excluded the *snowballstemmer* dependency, its two and only two versions being involved in the three configurations that pass all tests.
- We excluded the *request* dependency from this experiment. With 20 possible versions, this dependency would better be studied in a separate experiment.
- We limited the versions of *docutils* to the two newest versions, which we suspect to be incompatible with Sphinx.
- We limited the versions of the other dependencies to the 2 to 3 oldest versions, which we suspect to be no longer compatible with Sphinx.

This additional experiment, where we generated all possible environments, yielded 108 configurations. Each of those configurations failed 19 tests. This tends to confirm what the original experiment suggested: The newest versions of *docutils* and the oldest of other dependencies involved in the experiment are not compatible with the latest version of Sphinx. A complementary experiment (oldest version for *docutils*, newest versions for others), yielding 81 configurations, showed that all those configurations were able to pass all tests.

The 22 configurations generated by CAMP were able to verify our hypothesis. By analyzing those configurations, we devised two experiments, altogether including 189 configurations. Those configurations, a

very small subset of the 402 300 possible configurations, were able to provide knowledge that the developers of Sphinx could easily use to update the intervals of their implicit dependencies.

10 Conclusions

We have presented the CAMP tool, the STAMP approach to configuration test amplification. At the time of writing, CAMP includes 7 000 LoC of Python code, supported by more than 250 automated test-cases covering 88 % of the code. More than 10 persons have pushed together about 950 commits, yielding 30 releases, and closing 51 issues over the 68 opened so far.

Provided with a single test template and a model of its variation points, CAMP automatically generates alternative environments and gather evidence that the system under test withstands these variations. CAMP relies on the Z3 constraint solver to explore the space of possible configurations and on the Docker container engine to run each configuration in isolation. As we have shown on the Atom text editor and the Sphinx documentation generator, it is difficult to foresee the consequences of environmental changes.

As most automation efforts, engaging in configuration test amplification is an investment: While it takes time and effort to build the test template and its variability model, it pays back once tests run continuously throughout the life of the projects. Configuration tests thus helps avoid regression and preserve compatibility and portability.