Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Test them all, is it worth it? A ground truth comparison of configuration sampling strategies

Axel Halin · Alexandre Nuttinck · Mathieu Acher · Xavier Devroey · Gilles Perrouin · Benoit Baudry

Received: date / Accepted: date

Abstract Many approaches for testing configurable software systems start from the same assumption: it is impossible to test all configurations. This motivated the definition of variability-aware abstractions and sampling techniques to cope with large configuration spaces. Yet, there is no theoretical barrier that prevents the exhaustive testing of all configurations by simply enumerating them, if the effort required to do so remains acceptable. Not only this: we believe there is lots to be learned by systematically and exhaustively testing a configurable system.

In this article, we report on the first ever endeavor to test all possible configurations of an industry-strength, open source configurable software system, JHipster, a popular code generator for web applications. We built a testing scaffold for the 26,000+ configurations of JHipster using a cluster of 80 machines during 4 nights for a total of 4,376 hours (182 days) CPU time. We find that 35.70% configurations fail and we identify the feature interactions that cause the errors. We show that sampling testing strategies (like dissimilarity and 2-wise) (1) are more effective to find faults than the 12 default configurations used in the JHipster continuous integration; (2) can be too costly and exceed the available testing budget. We cross this quantitative analysis with the qualitative assessment of JHipster's lead developers.

Keywords Configuration sampling · variability-intensive system · software testing · JHipster · case study

A. Halin, A. Nuttinck, G. Perrouin (FNRS research associate)
PReCISE, Namur Digital Institute,
University of Namur, Belgium
E-mail: gilles.perrouin@unamur.be

M. Acher
IRISA, University of Rennes I, France
E-mail: mathieu.acher@irisa.fr

X. Devroey (orcid.org/0000-0002-0831-7606)
SERG, Delft University of Technology, The Netherlands
E-mail: x.d.m.devroey@tudelft.nl

B. Baudry
KTH Royal Institute of Technology, Sweden
E-mail: baudry@kth.se

1 Introduction

Configurable systems offer numerous options (or features) that promise to fit the needs of different users. New functionalities can be activated or deactivated while some technologies can be replaced by others for addressing a diversity of deployment contexts, usages, etc. The engineering of highly configurable systems is a standing goal of numerous software projects but it also has a significant cost in terms of development, maintenance, and testing. A major challenge for developers of configurable systems is to ensure that all combinations of options (configurations) correctly compile, build, and run. Configurations that fail can hurt potential users, miss opportunities, and degrade the success or reputation of a project. Though formal methods and program analysis can identify some classes of defects [9,48] – leading to variability-aware testing approaches (e.g., [28,29,34]) – a common practice is still to execute and *test* a sample of (representative) configurations. Indeed, enumerating all configurations is perceived as impossible, unpractical or both. While this is generally true, we believe there is lots to be learned by rigorously and exhaustively testing a configurable system. Prior empirical investigations (e.g., [32,42,44]) suggest that using a sample of configurations is effective to find configuration faults, at low cost. However, evaluations were carried out on a small subset of the total number of configurations or faults,

constituting a threat to validity. They typically rely on a corpus of faults that are mined from issue tracking systems. Knowing all the failures of the whole configurable system provides an unique opportunity to accurately assess the error-detection capabilities of sampling techniques *with a ground truth*. Another limitation of prior works is that the cost of testing configurations can only be estimated. They generally ignore the exact computational cost (e.g., time needed) or how difficult it is to instrument testing for any configuration.

This article aims to grow the body of knowledge (e.g., in the fields of combinatorial testing and software product line engineering [10, 19, 21, 32, 33, 44]) with a new research approach: the exhaustive testing of *all* configurations. We use JHipster, a popular code generator for web applications, as a case study. Our goals are: (i) to investigate the engineering effort and the computational resources needed for deriving and testing all configurations, and (ii) to discover how many failures and faults can be found using exhaustive testing in order to provide a ground truth for comparison of diverse testing strategies. We describe the efforts required to distribute the testing scaffold for the 26000+ configurations of JHipster, as well as the interaction bugs that we discovered. We cross this analysis with the qualitative assessment of JHipster's lead developers. Overall, we collect multiple sources that are of interest for (i) researchers interested in building evidence-based theories or tools for testing configurable systems; (ii) practitioners in charge of establishing a suitable strategy for testing their systems at each commit or release. This article builds on preliminary results [17] that introduced the JHipster case for research in configurable systems and described early experiments with the testing infrastructure on a very limited number of configurations (300). In addition to providing a quantitative assessment of sampling techniques on ***all*** the configurations, the present contribution presents numerous qualitative and quantitative insights on building the testing infrastructure itself and compare them with JHipster developers' current practice. In short, we report on the first ever endeavor to test all possible configurations of an industry-strength configurable software system. Specifically, the main contributions and findings of this article are:

1. a cost assessment and qualitative insights of engineering an infrastructure able to automatically test all configurations. This infrastructure is itself a configurable system and requires a substantial, error-prone, and iterative effort (8 man-month);
2. a computational cost assessment of testing all configurations using a cluster of distributed machines. Despite some optimizations, 4,376 hours (̃182 days) CPU time and 5.2 terabytes of free diskspace are needed to execute 26,257 configurations;
3. a quantitative and qualitative analysis of failures and faults. We found that 35.70% of all configurations fail: they either do not compile, cannot be built or fail to run. Six feature interactions (up to 4-wise) mostly explain this high percentage;
4. an assessment of sampling techniques. Dissimilarity and t-wise sampling techniques are effective to find faults that cause a lot of failures;
5. a retrospective analysis of JHipster practice. The 12 configurations used in the continuous integration for testing JHipster were not able to find the defects. It takes several weeks for the community to discover and fix the 6 faults;
6. a discussion on the future of JHipster testing based on collected evidence and feedback from JHipster's lead developers;
7. a feature model for JHipster v3.6.1 and a dataset to perform ground truth comparison of configuration sampling techniques, both available at `https://github.com/xdevroey/jhipster-dataset`.

The remainder of this article is organised as follows: section 2 presents the JHipster case study, the research questions, and methodology applied in this article; section 3 presents the human and computational cost of testing all JHipster configurations; section 4 presents the faults and failures found during JHipster testing; section 5 makes a ground truth comparison of the sampling strategies; section 6 gives the practitioners point of view on JHipster testing by presenting the results of our interview with JHipster developers; section 7 discusses the threats to validity; section 8 presents the related work; and section 9 wraps up with conclusions.

2 Case Study

JHipster is an open-source, industrially used generator for developing Web applications [22]. Started in 2013, the JHipster project has been increasingly popular (6000+ stars on Github) with a strong community of users and around 300 contributors in February 2017.

From a user-specified configuration, JHipster generates a complete technological stack constituted of Java and Spring Boot code (on the server side) and Angular and Bootstrap (on the front-end side). The generator supports several technologies ranging from the database used (e.g., *MySQL* or *MongoDB*), the authentication mechanism (e.g., *HTTP Session* or *Oauth2*), the support for social log-in (via existing social networks accounts), to the use of microservices. Technically, JHipster uses *npm* and *Bower* to manage dependencies and

Listing 1 Variability in _DatabaseConfiguration.java

```
(...)
@Configuration<% if (databaseType == 'sql') { %>
@EnableJpaRepositories("<%=packageName%>.repository")
@EnableJpaAuditing(...)
@EnableTransactionManagement<% } %>
(...)
public class DatabaseConfiguration
<% if (databaseType == 'mongodb') { %>
        extends AbstractMongoConfiguration
<% } %>{

 <%_ if (devDatabaseType == 'h2Disk' ||
     devDatabaseType == 'h2Memory') { _%>
   /**
   * Open the TCP port for the H2 database.
   * @return the H2 database TCP server
   * @throws SQLException if the server failed to
       start
   */
   @Bean(initMethod = "start", destroyMethod = "stop
       ")
   @Profile(Constants.SPRING_PROFILE_DEVELOPMENT)
   public Server h2TCPServer() throws SQLException {
      return Server.createTcpServer(...);
   }
  <%_ } _%>
(...)
```

Yeoman[1] (aka *yo*) tool to scaffold the application [41]. JHipster relies on conditional compilation with EJS[2] as a variability realisation mechanism. Listing 1 presents an excerpt of class *DatabaseConfiguration.java*. The options sql, mongodb, h2Disk, h2Memory operate over Java annotations, fields, methods, etc. For instance, on line 8, the inclusion of mongodb in a configuration means that DatabaseConfiguration will inherit from AbstractMongoConfiguration.

JHipster is a *complex* configurable system with the following characteristics: (i) a variety of languages (JavaScript, CSS, SQL, etc.) and advanced technologies (Maven, Docker, etc.) are combined to generate a variant of JHipster; (ii) there are 48 configuration options and a configurator guides users throughout different questions. Not all combinations of options are possible and there are 15 constraints between options; (iii) variability is scattered among numerous kinds of artifacts (*pom.xml*, Java classes, Docker files, etc.) and several options typically contribute to the activation or deactivation of portions of code, which is commonly observed in configurable software [23].

This complexity challenges core developers and contributors of JHipster. Unsurprisingly, numerous configuration faults have been reported on mailing lists and eventually fixed with commits[3]. Though formal methods and variability-aware program analysis can identify some defects [9,34,48], a significant effort would be needed to handle them in this technologically diverse stack. Thus, the current practice is rather to execute and test some configurations and JHipster offers opportunities to assess the cost and effectiveness of sampling strategies [10, 19, 21, 32, 33, 44]. Due to the reasonable number of options and the presence of 15 constraints, we (as researchers) also have an unique opportunity to gather a ground truth through the testing of *all* configurations.

[1] http://yeoman.io/

[2] http://www.embeddedjs.com/

[3] e.g., https://tinyurl.com/bugjhipster15

2.1 Research Questions

A first research question is to characterize the *cost* of such an exhaustive and automated testing strategy:

(RQ1.1) What is the cost of engineering an infrastructure capable of automatically deriving and testing all configurations?

(RQ1.2) What are the computational resources needed to test all configurations?

A second line of research is to qualify and quantify the configuration defects:

(RQ2.1) How many and what kinds of failures/faults can be found in all configurations?

In this article, we use the term *defect* to refer to either a fault or a failure. A *failure* is an "undesired effect observed in the system's delivered service" [31, 47] (e.g., the JHipster configuration fails to compile). We then consider that a *fault* is a cause of failures. As we found in our experiments (see Section 4), a single fault can explain many configuration failures since the same feature interactions cause the failure.

By collecting a *ground truth* (or reference) of defects, we can measure the effectiveness of sampling techniques. For example, is a random selection of 50 (says) configurations as effective to find failures/faults than an exhaustive testing? We can address this research question:

(RQ2.2) How effective are sampling techniques comparatively?

Finally, we can put in perspective the typical *trade-off* between the ability to find configuration defects and the cost of testing.

(RQ3) What is the most cost-effective sampling strategy? What are the recommendations for the JHipster project?

2.2 Methodology

We address these questions through quantitative and qualitative research. We initiated the work in Septem-

ber 2016 and selected JHipster 3.6.1[4] (release date: mid-August 2016). The 3.6.1 corrects a few bugs from 3.6.0; the choice of a "minor" release avoids finding bugs caused by an early and unstable release.

The two first authors worked full-time during 4 months to develop the infrastructure capable of testing all configurations of JHipster. They were graduate students, with strong skills in programming and computer science. Prior to the project's start, they have studied feature models and JHipster. We used GitHub to track the evolution of the testing infrastructure. We also performed numerous physical or virtual meetings (with Slack). Four other people have supervised the effort and provided guidance based on their expertise in software testing and software product line engineering. Through frequent exchanges, we gather several qualitative insights throughout the development.

Besides, we decided not to report faults whenever we found them. Indeed, we wanted to observe whether and how fast the JHipster community would discover and correct these faults. We monitored JHipster mailing lists to validate our testing infrastructure and characterize the configuration failures in a qualitative way. We have only considered GitHub issues since most of the JHipster activity is there. Additionally, we used statistical tools to quantify the number of defects, as well as to assess sampling techniques. Finally, we crossed our results with insights from three JHipster's lead developers.

3 All Configurations Testing Costs

We describe here our engineering efforts in building a fully automated testing infrastructure for all JHipster variants (**RQ1.1**). We also evaluate the computational cost of such an exhaustive testing (**RQ1.2**). We describe the necessary resources (man-power, time, machines) and report on encountered difficulties as well as lessons learned.

3.1 Reverse Engineering Variability

The first step towards a complete and thorough testing of JHipster variants is the modelling of its configuration space. JHipster comes with a command-line configurator. However, we quickly noticed that a brute force try of every possible combinations has scalability issues. Some answers activate or deactivate some questions and options. As a result, we rather considered the source code from GitHub for identifying options and constraints. Though options are scattered amongst artifacts, there is a central place that manages the configurator and then calls different sub-generators to derive a variant.

Listing 2 Configurator: server/prompt.js (excerpt)

```
(...)
when: function (response) {
    return applicationType === 'microservice';
  },
  type: 'list',
  name: 'databaseType',
  message: function (response) {
      return getNumberedQuestion('Which *type* of
          database would you like to use?',
          applicationType === 'microservice');},
  choices: [
      {value: 'no', name: 'No database'},
      {value: 'sql', name: 'SQL (H2, MySQL, MariaDB,
           PostgreSQL, Oracle)'},
      {value: 'mongodb', name: 'MongoDB'},
      {value: 'cassandra',name: 'Cassandra'}
  ],
  default: 1
(...)
```

We essentially consider *prompts.js*, which specifies questions prompted to the user during the configuration phase, possible answers (a.k.a. options), as well as constraints between the different options. Listing 2 gives an excerpt for the choice of a databaseType. Users can select no database, sql, mongodb, or cassandra options. There is a pre-condition stating that the prompt is presented only if the microservice option has been previously selected (in a previous question related to applicationType). In general, there are several conditions used for basically encoding constraints between options.

We modelled JHispter's variability using a feature model (e.g., [27]) to benefit from state-of-the-art reasoning techniques developed in software product line engineering [2, 4, 7, 8, 48]. Though there is a gap with the configurator specification (see Listing 2), we can encode its *configuration semantics* and hierarchically organize options with a feature model. We decided to interpret the meaning of the configurator as follows:

1. each multiple choice question is an (abstract) feature. In case of "yes" or "no" answer, questions are encoded as optional features (e.g., databaseType is optional in Listing 2);
2. each answer is a concrete feature (e.g., sql, mongodb, or cassandra in Listing 2). All answers to questions are exclusive and translated as alternative groups in the feature modeling jargon. A notable exception is the selection of testing frameworks in which several answers can be both selected; we translated them as an Or-group;

[4] https://github.com/jhipster/generator-jhipster/releases/tag/v3.6.1

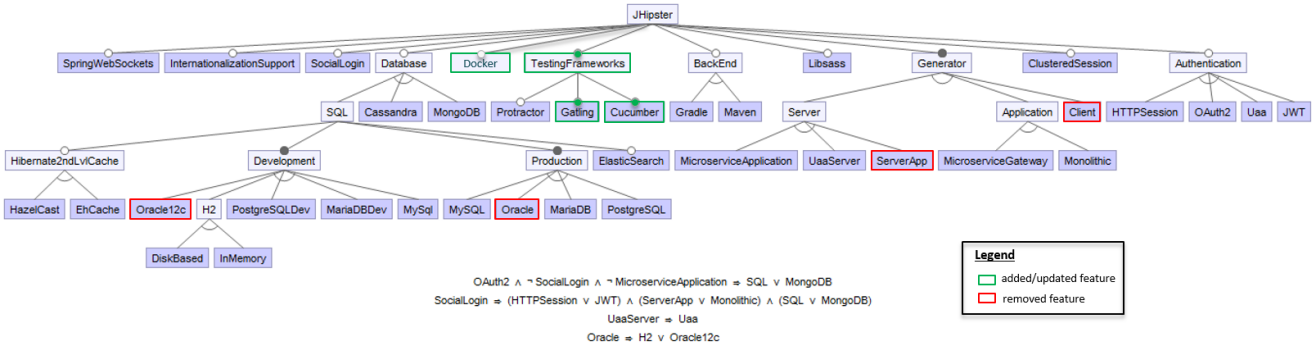


Fig. 1 JHipster reverse engineered feature model (only an excerpt of cross-tree constraints is given)

3. pre-conditions of questions are translated as constraints between features.

Based on an in-depth analysis of the source code and attempts with the configurator, we have manually reverse-engineered an initial feature model presented in Figure 1: 48 identified features and 15 constraints (we only present four of them in Figure 1 for the sake of clarity). The total number of valid configurations is 162,508.

Our goal was to derive and generate all JHipster variants corresponding to feature model configurations. Yet, we decided to adapt the initial model as follows:

1. we added Docker to build additional variants (i.e., we added a new optional feature Docker);
2. we excluded client/server standalones since there is a limited interest for users to consider the server (respectively client) without a client (respectively server) stack and failures most likely occur when both sides are inter-related;
3. we included the three testing frameworks in all variants. The three frameworks do not augment the functionality of JHipster and are typically here to improve the testing process, allowing us to gather as much information as possible about the variants;
4. we excluded Oracle-based variants. Oracle is a proprietary technology with technical specificities that are quite hard to fully automate (see next section).

Strictly speaking, we test *all* configurations of a *specialized* JHipster, essentially a JHipster without Oracle technology. Overall, we consider that our specialization of the feature model is conservative and still substantial. In the rest of this article, we are considering the original feature model of Figure 1 augmented with specialized constraints that negate red features (Oracle12c, Oracle, ServerApp, and Client) and that add green features and constraints (optional Docker and mandatory Gatling and Cucumber). This specialization leads to a total of **26,256** variants.

3.2 Fully Automated Derivation and Testing

From the feature model, we enumerated all valid configurations using solvers and FAMILIAR [2]. We developed a comprehensive workflow for testing each configuration. Figure 2 summarises the main steps (compilation, builds and tests). The first step is to synthesize a *.yo-rc.json* file from a feature model configuration. It allows us to skip the command-line questions-and-answers-based configurator; the command *yo jhipster* can directly use such a JSON file for launching the compilation of a variant. A monitoring of the whole testing process is performed to detect and log *failures* that can occur at several steps of the workflow. We faced several difficulties for instrumenting the workflow:

3.2.1 Engineering a configurable system for testing configurations

The execution of a unique and generic command for testing JHipster variants was not directly possible. For instance, the build of a JHipster application relies either on Maven or Gradle, two alternative features of our variability model. We developed variability-aware scripts to execute commands specific to a JHipster configuration. Command scripts include: starting database services, running database scripts (creation of tables, keyspaces, generation of entities, *etc.*), launching test commands, starting/stopping Docker, etc. As a concrete example, the inclusion of features h2 and Maven lead to the execution of the command: *"mvnw -Pdev"*; the choice of Gradle (instead of Maven) and mysql (instead of h2) in production mode would lead to the execution of another command: *"gradlew -Pprod"*. In total, 15 features of the original feature model influence (individually or through interactions with others) the way the testing workflow is executed. The first lessons learned are that (i) a non-trivial engineering effort is needed to build a *configuration-aware testing workflow* – testing a configurable system like JHipster requires to develop another

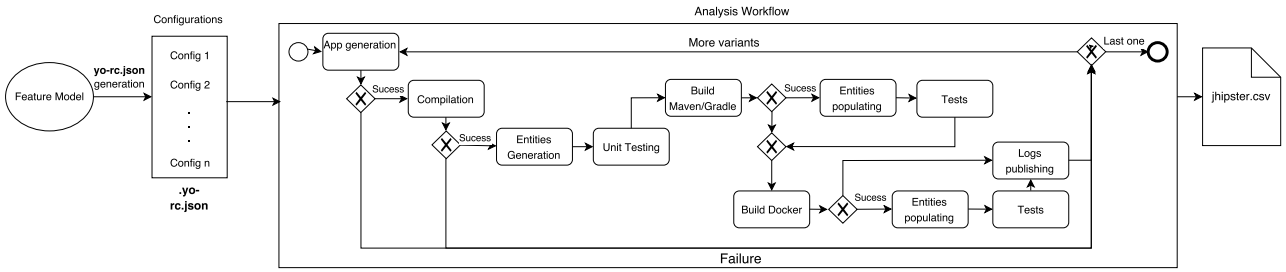


Fig. 2 Testing workflow of JHipster configurations/variants.

configurable system; (ii) the development was iterative and mainly consisted in *automating all tasks* originally considered as manual (e.g., starting database services).

3.2.2 Implementing testing procedures

After a successful build, we can execute and test a JHipster variant. A first challenge is to create the generic conditions (i.e., input data) under which all variants will be executed and tested. Technically, we need to populate Web applications with *entities* (i.e., structured data like tables in a SQL database or documents in MongoDB for instance) to test both the server-side (in charge of accessing and storing data) and the client-side (in charge of presenting data). JHipster entities are created using a domain-specific language called JDL, close to UML class diagram formalism. We decided to reuse the entity model template given by the JHipster team[5]. We created 3 entity models for MongoDB, Cassandra, and "others" because some database technologies vary in terms of JDL expressiveness they can support (e.g., you cannot have relationships between entities with a MongoDB database). After entities creation with JDL, we run several tests: integration tests with *Spring Test Context* framework, U.I tests with *Karma.js*, etc. We instantiate these entities using the Web user interface through Selenium scripts. We integrate the following testing frameworks to compute additional metrics: Cucumber, Gatling and Protractor. We also implement generic oracles that analyse and extract log error messages.

Finding commonalities among the testing procedures participates to the engineering of a configuration-aware testing infrastructure. The major difficulty was to develop input data (entities) and test cases (e.g., Selenium scripts) that are *generic* and can be applied to all JHipster variants.

3.2.3 Building an all-inclusive testing environment

Each JHipster configuration requires to use specific tools and pre-defined settings. Without them, the compilation, build, or execution cannot be performed. A substantial engineering effort was needed to build an integrated environment capable of deriving any JHipster configuration. The concrete result is a Debian image with all tools pre-installed and pre-configured. This process was based on numerous tries and errors, using some configurations. At the end, we converged on an all-inclusive environment.

3.2.4 Distributing the tests

The number of JHipster variants led us to consider strategies to scale up the execution of the testing workflow. We decided to rely on Grid'5000[6], a large-scale testbed offering a large amount of computational resources [5]. We used numerous distributed machines, each in charge of testing a subset of configurations. Small scale experiments (e.g., on local machines) helped us to manage distribution issues in an incremental way. Distributing the computation further motivated our previous needs of testing automation and preset Debian images.

3.2.5 Opportunistic optimizations and sharing

Each JHipster configuration requires to download numerous Java and JavaScript dependencies, which consumes bandwidth and increases JHipster variant generation time. To optimise this in a distributed setting, we downloaded all possible Maven, npm and Bower dependencies – once and for all configurations. We eventually obtained a Maven cache of 481Mo and a node_modules (for JavaScript dependencies) of 249Mo. Furthermore, we build a Docker variant right after the classical build (see Figure 2) to derive two JHipster variants (with and without Docker) without restarting the whole derivation process.

[5] `https://jhipster.github.io/jdl-studio/`

[6] `https://www.grid5000.fr`

3.2.6 Validation of the testing infrastructure

A recurring reaction after a failed build was to wonder whether the failure was due to a buggy JHipster variant or an invalid assumption/configuration of our infrastructure. We extensively tried some selected configurations for which we know it should work and some for which we know it should not work. Based on some potential failures, we reproduced them on a local machine and studied the error messages. We also used statistical methods and GitHub issues to validate some of the failures (see next Section). This co-validation, though difficult, was necessary to gain confidence in our infrastructure. After numerous tries on our selected configurations, we launched the testing workflow for all the configurations (selected ones included).

3.3 Human Cost

The development of the complete derivation and testing infrastructure was achieved in about 4 months by 2 people (i.e., **8 person/month** in total). For each activity, we report the duration of the effort realized in the first place. Some modifications were also made in parallel to improve different parts of the solution – we count this duration in subsequent activities.

3.3.1 Modelling configurations

The elaboration of the first major version of the feature model took us about **2 weeks** based on the analysis of the JHipster code and configurator.

3.3.2 Configuration-aware testing workflow

Based on the feature model, we initiated the development of the testing workflow. We added features and testing procedures in an incremental way. The effort spanned on a period of **8 weeks**.

3.3.3 All-inclusive environment

The building of the Debian image was done in parallel to the testing workflow. It also lasted a period of **8 weeks** for identifying all possible tools and settings needed.

3.3.4 Distributing the computation

We decided to deploy on Grid'5000 at the end of November and the implementation has lasted **6 weeks**. It includes a learning phase (1 week), the optimization for caching dependencies, and the gathering of results in a central place (a CSV-like table with logs).

3.4 Computational Cost

We used a network of machines that allowed us to test all 26,256 configurations in *less than a week*. Specifically, we performed a reservation of **80** machines for **4 periods** (4 nights) of **13 hours**. The analysis of 6 configurations took on average about 60 minutes. The total CPU time of the workflow on all the configurations is **4,376 hours**. Besides CPU time, the processing of all variants also required enough free disk space. Each scaffolded Web application occupies between 400MB and 450MB, thus forming a total of **5.2 terabytes**.

We replicated three times our exhaustive analysis (with minor modifications of our testing procedure each time); we found similar numbers for assessing the computational cost on Grid'5000. As part of our last experiment, we observed suspicious failures for 2325 configurations with the same error message ("Communications link failure"). A new run of these configurations yielded consistent results.

> The testing infrastructure is itself a configurable system and requires a substantial engineering effort (8 man-months) to cover all design, implementation and validation activities, the latter being the most difficult. Testing all configurations requires a significant amount of computational resources (4376 hours CPU time and 5.2 terabytes of disk space).

4 Results of the testing workflow execution

The execution of the testing workflow yielded a large file comprising numerous results for each configuration. This file[7] allows to identify failing configurations, i.e., configurations that do not compile or build. In addition, we also exploited stack traces for grouping together some failures. We present here the ratios of failures and associated faults (**RQ2.1**).

4.1 Bugs: A Quick Inventory

Out of the **26,256** configurations we tested, we found that **9,376** (**35.70**%) failed. This failure occurred either at compile time or during build time although the generation was successful. We also found that some features were more concerned by failures as depicted in Figure 3. Regarding the application type, for instance, *microservice gateways* and *microservice applications* are

[7] Complete results are available at `https://github.com/xdevroey/jhipster-dataset/tree/master/v3.6.1`.

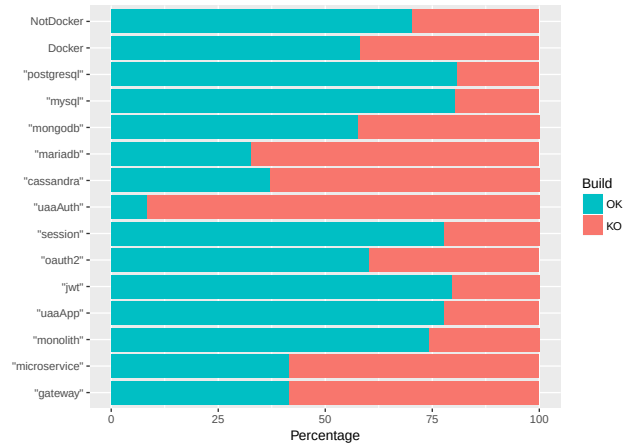


Fig. 3 Proportion of build failure by feature

proportionally more impacted than *monolithic applications* or *UAA server* with, respectively, **58.37**% of failures (4,184 failing microservice gateways configurations) and **58.3**% of failures (532 failing microservice applications configurations). *UAA authentication* is involved in most of the failures: **91.66**% of *UAA-based microservices* applications (4,114 configurations) fail to deploy.

4.2 Statistical Analysis

Previous results do not show the root causes of the configuration failures – what features or interactions between features are involved in the failures? To investigate correlations between features and the failure results, we decided to use the Association Rule learning method [16]. It aims at extracting relations between variables of large data-sets. The outputs are a set of rules, each constituted by: *Left-hand side (LHS)*, the antecedent of the rule; *Right-hand side (RHS)*, the consequent of the rule; *Support*, the proportion of configurations where LHS holds; *Confidence*, the proportion of configurations where RHS holds.

Table 1 gives some examples. We parametrized the method as follows. First, we restrained ourselves to rules where the RHS was either *Build=KO* or *Compile=KO*. Second, we fixed the confidence to 1: if a rule has a confidence below 1 then it is not asserted in all configurations where the LHS expression holds – the failure does not occur in all cases. Third, we lowered the support in order to catch all failures, even those afflicting smaller proportion of the configurations. For instance, only 224 configurations fail due to a compilation error; in spite of a low support, we can still extract rules for which the RHS is *Compile=KO*. We computed redundant rules using facilities of the R package ARULES[8]. As some association rules can contain already known constraints of the feature model, we ignored some of them.

We first considered association rules for which the size of the LHS is either 1, 2 or 3. We extracted 5 different rules involving two features (see Table 1). We found no rule involving 1 or 3 features. We decided to have a look at the 200 association rules for which the LHS is of size 4. We found out a sixth association rule that incidentally corresponds to one of the first failures we encountered in the early stages of this study.

Table 1 shows that there is only one rule with the RHS being *Compile=KO*. According to this rule, all configurations in which the database is MongoDB and social login feature is enabled (128 configurations) fail to compile. The other 5 rules are related to a build failure. Figure 4 reports on the proportion of failed configurations that include the LHS of each association rule. Such LHS can be seen as a feature interaction fault that causes failures. For example, the combination of MariaDB and Gradle explains *37%* of failed configurations (or 13% of all configurations). We conclude that six feature interaction faults explain 98.65% of the failures.

4.3 Qualitative Analysis

We now characterize the 6 important faults, caused by the interactions of several features (between 2 features and 4 features). Table 1 gives the support, confidence for each association rule. We also confirm each fault by giving the GitHub issue and date of fix.

4.3.1 MariaDB with Docker

This fault is the only one caused by the interaction of 4 features: it concerns *monolithic* web-applications relying on MariaDB as production database, where the search-engine (ElasticSearch) is disabled and built with Docker. These variants amount to **1,468** configurations and the root cause of this bug lies in the template file *src/main/docker/_app.yml* where a condition (*if prodDB = MariaDB*) is missing.

4.3.2 MariaDB using Gradle

This second fault concerns variants relying on Gradle as build tool and MariaDB as the database (**3,519** configurations). It is caused by a missing dependency in template file *server/template/gradle/_liquibase.gradle*.

[8] `https://cran.r-project.org/web/packages/arules/index.html`

Table 1 Association rules involving compilation and build failures

Left-hand side	Right-hand side	Support	Conf.	GitHub Issue	Report/Correction date
DatabaseType="mongodb", EnableSocialSignIn=true	Compile=KO	0.488 %	1	4037	27 Aug 2016 (report and fix for milestone 3.7.0)
prodDatabaseType="mariadb", buildTool="gradle"	Build=KO	16.179 %	1	4222	27 Sep 2016 (report and fix for milestone 3.9.0)
Docker=true, authenticationType="uaa"	Build=KO	6.825 %	1	UAA is in Beta	Not corrected
authenticationType="uaa", hibernateCache="no"	Build=KO	2.438 %	1	4225	28 Sep 2016 (report and fix for milestone 3.9.0)
authenticationType="uaa", hibernateCache="ehcache"	Build=KO	2.194 %	1	4225	28 Sep 2016 (report and fix for milestone 3.9.0)
prodDatabaseType="mariadb", applicationType="monolith", searchEngine="false", Docker="true"	Build=KO	5.590%	1	4543	24 Nov 2016 (report and fix for milestone 3.12.0)

4.3.3 UAA authentication with Docker

The third fault occurs in Microservice Gateways or Microservice applications using an UAA server as authentication mechanism (**1,703** Web apps). This bug is encountered at build time, with Docker, and it is due to the absence of UAA server Docker image. It is a known issue but it has not been corrected yet, UAA servers are still in beta versions.

4.3.4 UAA authentication with Ehcache as Hibernate 2nd level cache

This fourth fault concerns Microservice Gateways and Microservice applications, using a UAA authentication mechanism. When deploying manually (i.e., with Maven or Gradle), the web application is unable to reach the deployed UAA instance. This bug seems to be related to the selection of Hibernate cache and impacts **1,667** configurations.

4.3.5 OAuth2 authentication with SQL database

This defect is faced **649** times, when trying to deploy a web-app, using a SQL database (Mysql, PostgreSQL or MariaDB) and an OAuth2 authentication, with Docker. It was reported on August 20th, 2016 but the JHipster team was unable to reproduce it on their end.

4.3.6 Social Login with MongoDB

This sixth fault is the only one occurring at compile time. Combining MongoDB and social login leads to **128** configurations that fail. The source of this issue is a missing import in class *SocialUserConnection.java*. This import is not in a conditional compilation condition in the template file while it should be.

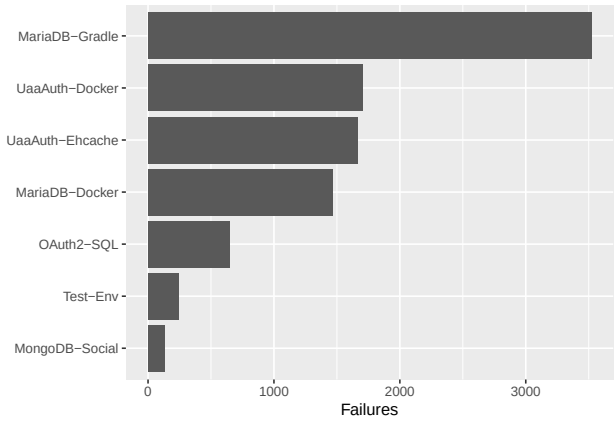


Fig. 4 Proportion of failures by fault

4.3.7 Testing infrastructure

We have not found a common fault for the remaining **242** configurations that fail. We came to this conclusion after a thorough and manual investigation of all logs. (Such configurations are tagged by "ISSUE:env" in the column "bug" of the JHipster results CSV file available online `https://github.com/xdevroey/jhipster-dataset`.) We noticed that, despite our validation effort with the infrastructure (see **RQ1**), the observed failures are caused by the testing tools and environment. Specifically, the causes of the failures can be categorized in two groups: *(i)* several network access issues in the grid that can affect the testing workflow at any stage and *(ii)* several unidentified errors in the configuration of building tools (gulp in our case).

5 Sampling techniques comparison

In this section, we discuss the sampling strategy used by the JHipster team and then use our dataset to make a

ground truth comparison of six state of the art sampling techniques (**RQ2.2**).

5.1 JHipster team sampling strategy

The JHipster team uses a sample of twelve representative configurations for the version 3.6.1, to test their generator (see Section 6.1 for the explanation on how these were sampled). During a period of several weeks, the testing configurations have been used at *each commit* (see also Section 6.1). These configurations fail to reveal any problem, i.e., the Web-applications corresponding to the configurations successfully compiled, build and run. We assessed these configurations with our own testing infrastructure and came to the same observation. We thus conclude that this sample was not effective to reveal defects.

5.2 Comparison with t-wise/random sampling

As testing all configurations is very costly (see **RQ1.2**), sampling techniques remain of interest. Ideally, we would like to find a maximum of failures and faults with a minimum of configurations in the sampling. For each failure, we associate a fault through the automatic analysis of features involved in the failed configuration (see previous subsections).

We address **RQ2.2** with numerous sampling techniques considered in the literature [1, 25, 32, 39]. For each technique, we report on the number of failures and faults.

5.2.1 Sampling techniques

t-wise sampling. We selected 4 variations of the t-wise criteria: **1-wise**, **2-wise**, **3-wise** and **4-wise**. We generate the samples with *SPLCAT* [25]. This sampling technique selects a subset of the configuration space such as each interaction of t features is covered [39]. The 4 variations yield samples of respectively **8**, **41**, **126** and **374** configurations. **1-wise** only finds **2** faults; **2-wise** discovers **5 out of 6 faults**; **3-wise** and **4-wise** find **all** of them.

One-disabled sampling. The core idea of **one-disabled** algorithm is to extract configurations in which all features are activated but one [1, 32]. In our case, we can select each configuration where *Docker* is disabled and *SpringWebSockets*, *InternationalizationSupport*, *SocialLogin*, *Libsass* and *ClusteredSession* are enabled; the process is then repeated for all other combinations of features. This criteria allows various strategies regarding its implementation. For each disabled feature, Medeiros *et al.* [32] consider the first valid configuration returned by the solver. But there may be several valid configurations for a given disabled feature. Since SAT solvers relies on internal orders to process solutions (see [19]) the first valid solution will always be the same. The good point is that it makes the algorithm deterministic. However, it implicitly links the bug-finding ability of the algorithm with the solver's internal order and to the best of our knowledge, there is no reason why it should be linked. To overcome any bias in selecting "the first" valid configuration, we decided to apply a random selection. We therefore select a valid random configuration for each disabled feature (called **one-disabled** in our results) and repeat experiments to get significant results. This gives us a sample of **34 configurations** which detects on average **2.4 faults** out of 6.
Additionally, we also retain all valid configurations where one feature is disabled and the other are enabled (called **all-one-disabled** in our results). The all-one-disabled sampling yields a total sample of **922 configurations** that identifies **all faults but one**.

One-enabled and most-enabled-disabled sampling. In the same way, we implemented sampling algorithms covering the **one-enabled** and **most-enabled-disabled** criteria [1, 32]. The former mirrors one-disabled and consists in enabling each feature one at a time. The latter selects all configurations where most of the features are selected and most of the features are deselected. In any case, we took care of only considering configurations that are valid w.r.t. the constraints of the feature model. As for one-disabled, we choose to randomly select valid configurations instead of taking the first one returned by the solver. One-enabled extracts a sample of **34 configurations** which detects **3.15 faults** on average. And Most-enabled-disabled gives a sample of **2 configurations** that detects **0.67 faults** on average. Considering all valid configurations, **all-one-enabled** extracts a sample of **2340 configurations** and identifies **every major fault**. **All-most-enabled-disabled** gives a sample of **574 configurations** that identifies **2 faults** out of 6.

Dissimilarity sampling. We also considered *dissimilarity* testing for software product lines [3,19], using a tool called **PLEDGE** [20]. This technique approximates t-wise coverage by generating dissimilar configurations (in terms of shared features amongst these configurations). From a set of random configurations of a specified cardinality, an evolutionary algorithm evolves this set such that the distances amongst configuration are

Table 2 Efficiency of different sampling techniques

Sampling technique	Sample size	Failures (σ)	Failures eff.	Faults (σ)	Fault eff.
Random(8)	8	2.857 (1.313)	35.71%	2.180 (0.978)	27.25%
PLEDGE(8)	8	3.160 (1.230)	39.50%	2.140 (0.825)	26.75%
1-wise	8	2.000 (N.A.)	25.00%	2.000 (N.A.)	25.00%
Random(12)	12	4.285 (1.790)	35.71%	2.700 (1.040)	22.5%
PLEDGE(12)	12	4.920 (1.230)	41.00%	2.820 (0.909)	23.50%
2-wise	41	14.000 (N.A.)	34.15%	5.000 (N.A.)	12.20%
Random(41)	41	14.641 (3.182)	35.71%	4.490 (0.718)	10.95%
PLEDGE(41)	41	17.640 (2.500)	43.02%	4.700 (0.831)	11.46%
3-wise	126	52.000 (N.A.)	41.27%	6.000 (N.A.)	4.76%
Random(126)	126	44.995 (4.911)	35.71%	5.280 (0.533)	4.19%
PLEDGE(126)	126	49.080 (11.581)	38.95%	4.660 (0.698)	3.70%
Random(374)	374	133.555 (8.406)	35.71%	5.580 (0.496)	1.49%
PLEDGE(374)	374	139.200 (31.797)	37.17%	4.620 (1.181)	1.24%
4-wise	374	161.000 (N.A.)	43.05%	6.000 (N.A.)	1.60%
Most-enabled-disabled	2	0.683 (0.622)	34.15%	0.670 (0.614)	33.50%
All-most-enabled-disabled	574	190.000 (N.A.)	33.10%	2.000 (N.A.)	0.35%
One-disabled	34	7.699 (2.204)	0.23%	2.398 (0.878)	0.07%
All-one-disabled	922	253.000 (N.A.)	27.44%	5.000 (N.A.)	0.54%
One-enabled	34	12.508 (2.660)	0.37%	3.147 (0.698)	0.09%
All-one-enabled	2,340	872.000 (N.A.)	37.26%	6.000 (N.A.)	0.26%
ALL	26,256	9,376.000 (N.A.)	35.71%	6.000 (N.A.)	0.02%

maximal – by replacing a configuration at each iteration – within a certain amount of time. We retained this technique because it can afford any testing budget (sample size and generation time). For each sample size, we report the average failures and faults for 100 PLEDGE executions with the greedy method in 60 secs [20]. We selected (respectively) **8**, **12**, **41**, **126** and **374** configurations, finding (respectively) **2.14**, **2.82**, **4.70**, **4.66** and **4.60** faults out of 6.

Random sampling. Finally, we considered **random** samples from size 1 to 2500. The random samples exhibit, by construction, 35.71% of failures on average (the same percentage that is in the whole dataset). To compute the number of unique faults, we simulated 100 random selections. We find, on average, respectively **2.18**, **2.7**, **4.49**, **5.28** and **5.58** faults for respectively **8**, **12**, **41**, **126** and **374** configurations.

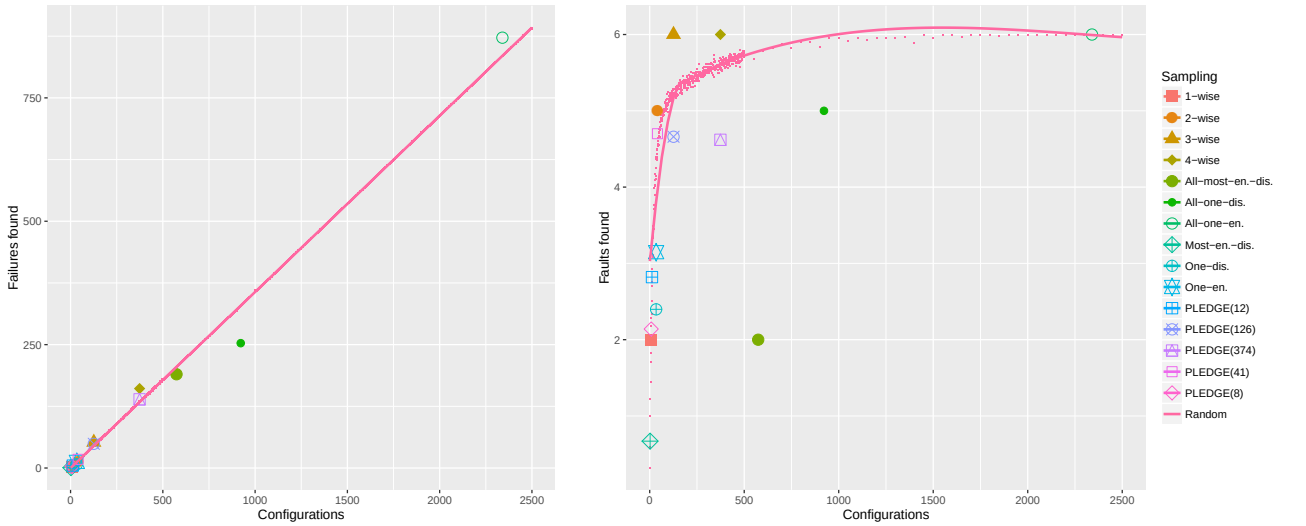
5.2.2 Fault and failure efficiency

We consider two main metrics to compare the efficiency of sampling techniques to find faults and failures w.r.t the sample size. *Failure efficiency* is the ratio of *failures to sample size*. *Fault efficiency* is the ratio of *faults to sample size*.

The results are summarized in Table 2. We present in Figure 5(a) (respectively, Figure 5(b)) the evolution of *failures* (respectively, *faults*) w.r.t. the size of random samples. To ease comparison, we place reference points corresponding to results of other sampling techniques. A first observation is that random is a strong baseline for both failures and faults. 2-wise or 3-wise sampling techniques are slightly more efficient to identify faults than random. On the contrary, all-one-enabled, one-enabled, all-one-disabled, one-disabled and all-most-enabled-disabled identify less faults than random samples of the same size. Most-enabled-disabled is efficient on average to detect faults (33.5% on average) but requires to be "lucky". In particular, the first configurations returned by the solver (as done in [32]) discovered 0 fault. This shows the sensitivity of the selection strategy amongst valid configurations matching the most-enabled-disabled criterion. Based on our experience, we recommend researchers the use of a random strategy instead of picking the first configurations when assessing one-disabled, one-enabled, and most-enabled-disabled.

PLEDGE is superior to random for small sample sizes. The significant difference between 2-wise and 3-wise is explained by the sample size: although the latter finds all the bugs (one more than 2-wise) its sample size is triple (126 configurations against 41 for 2-wise). In general, a relatively small sample is sufficient to quickly identify the 5 or 6 most important faults – there is no need to cover the whole configuration space.

A second observation is that there is no correlation between failure efficiency and fault efficiency. For example, all-one-enabled has a failure efficiency of 37.26% (better than random and many techniques) but is one of the worst technique in terms of fault rate due of its high sample size. In addition, some techniques, like all-most-enable-disabled, can find numerous failures that in fact correspond to the same fault.



(a) Failures found by sampling techniques

(b) Faults found by sampling techniques

Fig. 5 Defects found by sampling techniques

5.2.3 Discussion

Our results show that the choice of a metric (failure-detection or fault-detection capability) can largely influence the choice of a sampling technique. Our initial assumption was that the detection of one failure leads to the finding of the associated fault. The GitHub reports and our qualitative analysis show that it is indeed the case in JHipster: contributors can easily find the root causes based on a *manual* analysis of a configuration failure. In other practical contexts, finding the faulty features or feature interactions can be much more difficult and less immediate. It can be beneficial to *replicate* a fault with many failures and then use statistical methods (such as association rules) to identify feature interactions always involved in the fault. As a result, the ability of finding failures may be more important than in JHipster case. A tradeoff between failure and fault efficiency can certainly be considered when choosing the sampling technique.

Exhaustive testing shows that almost 36% of the configurations fail. Our analysis identifies 6 interaction faults as the root cause for this high percentage.

Exhaustive testing also sheds a new light on sampling techniques: (i) the 12 configurations used by the JHipster team do not find any defect; (ii) yet, 41 configurations are sufficient to cover the 5 most important faults; (iii) dissimilarity and t-wise sampling are the most effective; (iv) there is no correlation between failure and fault efficiencies.

6 Practitioners Viewpoint

We interviewed the JHipster lead developer, Julien Dubois, during one hour and a half, at the end of January. We prepared a set of questions and performed a semi-structured interview on Skype for allowing new ideas during the meeting. We then exchanged emails with two core developers of JHipster, Deepu K Sasidharan and Pascal Grimaud. Based on an early draft of our article, they clarified some points and freely reacted to some of our recommendations. We wanted to get insights on how JHipster was developped, used, and tested. We also aimed to confront our empirical results with their current practice.

6.1 Jhipster's testing strategy

6.1.1 Continuous testing

JHipster relies on a continuous integration platform (Travis) integrated into GitHub. At the time of the

release 3.6.1, the free installation of Travis allowed to perform 5 different builds in parallel, at each commit. JHipster exploits this feature to only test 12 configurations. JHipster developers give the following explanations: *"The only limit was that you can only run 5 concurrent jobs so having more options would take more time to run the CI and hence affect our turn around hence we decided on a practical limit on the number [...] We only test the 12 combinations because we focus on most popular options and leave the less popular options out."* Julien also mentioned that his company IPPON provides some machines used to perform additional tests. We can consider that the testing budget of JHipster 3.6.1 was limited to 12 configurations. It has a strong implication on our empirical results: *Despite their effectiveness, some sampling strategies we have considered exceed the available testing budget of the project.* For example, a 2-wise sample has 41 configurations and is not adequate. A remaining solution is dissimilarity sampling (PLEDGE) of 12 configurations, capable of finding 5 failures and 3 faults.

6.1.2 Sampling strategy

How have these 12 configurations been selected? According to Julien, it is both based on *intimate technical knowledge of the technologies* and *a statistical prioritization approach*. Specifically, when a given JHipster installation is configured, the user can send anonymous data to the the JHipster team so that it is possible to obtain a partial view on the configurations installed. The most popular features have been retained to choose the 12 configurations. For example, this may partly explain that configurations with Gradle are more buggy than those with Maven – we learned that Gradle is used in less than 20% of installations. There were also some discussions about improving the maintenance of Gradle, due to its popularity within a subset of contributors. The prioritization of popular configurations is perfectly understandable. Such a sample has the merit of ensuring that, at each commit, popular combinations of features are still valid (acting as non-regression tests). However, corner cases and some feature interactions are not covered, possibly leading to high percentage of failures.

6.2 Merits and limits of exhaustive testing

Julien welcomed the initiative and was seriously impressed by the unprecedented engineering effort and the 36% failures. We asked whether the version 3.6.1 had special properties, perhaps explaining the 36% of failures. He refuted this assumption and rather stated that the JHipster version was a major and stable release. We explained that most of the defects we found were reported by the JHipster community. The lead developer was aware of some interactions that caused problems in JHipster. These are known mostly from experience and not via the application of a systematic process. However, he ignored the significance of the failures. The high percentage of failures we found should be seriously considered since a significant number of users may be impacted given the popularity of the project. Even if faults involve rarely used configurations, he considered that the strength of JHipster is precisely to offer a diverse set of technologies. The effort of finding many failures and faults is therefore highly valuable.

We then discussed the limits of testing all configurations. The cost of building a grid/cluster infrastructure is currently out of reach for the JHipster open-source project, due to the current lack of investments. JHipster developers stated: *"even if we had limitless testing infrastructure, I do not think we will ever test out all possible options due to the time it would take"*. This observation is not in contradiction with our research method. Our goal was not to promote an exhaustive testing of JHipster but rather to investigate a cost-effective strategy based on collected evidence.

Another important insight is that *"the testing budget was more based on the time it would take and the resource it would use on a free infrastructure. If we let each continuous integration build to run for few hours then we would have to wait that long to merge pull request and to make releases etc. So it adds up lag affecting our ability to release quickly and add features and fixes quickly. So turn around IMO is something you need to consider for continuous integration"*.

Finally, Julien mentioned an initiative[9] to build an all-inclusive environment capable of hosting any configuration. It is for JHipster developers and aims to ease the testing of a JHipster configuration on a local machine. In our case, we built a similar environment with the additional objective of automating the test of configurations. We have also validated this environment for all configurations in a distributed setting.

6.3 Discussions

On the basis of multiple collected insights, we discuss tradeoffs to consider when testing JHipster and address **RQ3**.

[9] `https://github.com/jhipster/jhipster-devbox`

6.3.1 Sampling strategy

Our empirical results suggest to use a dissimilarity sampling strategy in replacement to the current sampling based on statistical prioritization. It is one of the most effective strategy for finding failures and faults and it does not exceed the budget. In general, the focus should be on covering as much feature interactions as possible. If the testing budget can be sufficiently increased, t-wise strategies can be considered as well. However, developers remind us that *"from a practical standpoint, a random sampling has possibility of us missing an issue in a very popular option thus causing huge impact, forcing us to make emergency releases etc, where as missing issues in a rarely used option does not have that implication"*. This applies to t-wise and dissimilarity techniques as well. Hence, one should find a tradeoff between cost, popularity, and effectiveness of sampling techniques. We see this as an opportunity to further experiment with multi-objective techniques [18, 37, 45].

6.3.2 Sampling size

Our empirical results and discussions with JHipster developers suggest that the testing budget was simply too low for JHipster 3.6.1, especially when popular configurations are included in the sampling. According to JHipster developers, the testing budget *"has increased to 19 now with JHipster 4, and we also have additional batch jobs running daily tripling the number of combinations [...] We settled on 19 configurations to keep build times within acceptable limits"*[10].

An ambitious and long-term objective is to crowdsource the testing effort with contributors. Users can lend their machines for testing some JHipster configurations while a subset of developers could also be involved with the help of dedicated machines. In complement to continuous testing of some popular configurations, a parallel effort could be made to seek failures (if any) on a diversified set of configurations, possibly less popular.

6.3.3 Configuration-aware testing infrastructure

In any case, we recommend to develop and *maintain* a configuration-aware testing infrastructure. Without a ready-to-use environment, contributors will not be able to help in testing configurations. It is also pointless to increase the sample if there is no automated procedure capable of processing the constituted configurations. The major challenge will be to follow the evolution of JHipster and make the testing tractable. A formal model of the configurator should be extracted for logically reasoning and implementing random or t-wise sampling. New or modified features of JHipster should be handled in the testing workflow; they can also have an impact on the tools and packages needed to instrument the process.

[10] Discussions are available at `https://github.com/jhipster/generator-jhipster/issues/4301`

7 Threats to Validity

Our engineering effort has focused on a single but industrial and complex system. We expect more insights into characteristics of real-world systems than using diverse but smaller or synthetic benchmarks. With the possible consideration of all JHipster configurations, we gain a ground truth that allows us to precisely assess sampling techniques.

Threats to internal validity are mainly related to the quality of our testing infrastructure. An error in the feature model or in the configuration-aware testing workflow can typically produce wrong failures. As reported, the validation of our solution has been a major concern during 8 man-months of development. We have used several strategies, from statistical computations to manual reviews of individual failures to mitigate this threat. We found all faults reported by the JHipster community and new failures.

For the other remaining 242 configurations that fail due to our test infrastructure (see Section 4.3.7), there might be false positives. Since they only represent 0.9% of all JHipster configurations, such false positives would have a marginal incidence on the results. In fact, this situation is likely to happen in a real (continuous integration) distributed testing environment (e.g., as reported in [6]). We thus decided to keep those configurations in the dataset. Additionally, they can serve as a baseline to improve our testing infrastructure for the next versions of Jhipster.

8 Related Work

The difficulty to assess all the possible configurations in the general case lead to the development of various sampling techniques, differing by their coverage criteria (e.g., pair-wise [10, 38, 49] or dissimilarity amongst configurations [19]), use of exact algorithms [21, 24], or metaheuristics [14, 19, 37, 45].

8.1 Comparison of sampling strategies

Perrouin *et al.* [38] compared two exact approaches on five feature models of the SPLOT repository w.r.t to

performance of t-wise generation and configuration diversity. Hervieu *et al.* [21] also used models from the SPLOT repository to produce a small number of configurations. Empirical investigations were pursued on larger models (1,000 features and above) notably on OS kernels (e.g., [19, 25]) demonstrating the relevance of metaheuristics for large sampling tasks [18,35]. However, these comparisons were performed at the model level (there was no product associated to these models) using artificial faults.

Several authors considered sampling on actual systems, thus dealing with real faults. Johansen *et al.* [26] extended SPLCAT by adding weights on sub-product lines and apply their approach to the Eclipse IDE and to TOMRA, an industrial product line. Steffens *et al.* [36] applied the Moso-Polite pairwise tool on an electronic module allowing 432 configurations to derive metrics regarding the test reduction effort. Additionally, they also exhibited a few cases where an higher interaction strength was required (3-wise). Sanchez *et al.* [43] modelled a subset of the Drupal framework and (manually) examined how faults (extracted from the Drupal's issue tracking system) were related to feature interactions: Amongst 390 faults, 11 were related to 2-wise interactions, 1 to 3-wise interactions, and the rest was related to single features. A further analysis on this case revealed that with respect to 3392 faults, 160 were related to up to 4-wise interactions [42]. Medeiros *et al.* [32] compared 10 sampling algorithms on a corpus of existing configurations faults taken from a large set of configurable systems. The authors found that 2-wise interactions covered a large subset of faults (as it is commonly assumed) but also made the case for higher-interaction strengths (seven in their study) and that sampling algorithms such as most-enabled-disabled are most efficient in that case. Our study adds more empirical evidence on the fact that higher-interaction is desirable to find all bugs.

Despite the number of empirical investigations (e.g., [13, 40]) and surveys (e.g., [11, 12, 48]) to compare such approaches, many focused on subsets to make the analyses tractable. Being able to execute all configurations led us to consider actual failures and collect a ground truth. It helps to gather insights for better understanding the interactions in large configuration spaces [33, 49].

8.2 Testing at Scale

Greiler *et al.* [15] investigated how developers tested their plug-ins in the Eclipse framework. The prevalent technique used was unit testing of individual plug-ins without any systematic approach to handle combination with other plugins. Interviewed people complained about the difficulty to set up a proper integration environment, long execution times (few minutes), and lack of best practices. In the case of JHipster, we reported on similar difficulties with the testing environment. Furthermore, execution time prevailed on the testing budget: it took 1h30 to build and test 12 configurations. However, the JHipster team is eager to try quicker ways for building of configurations and for the last release (4.0), they extended the budget to 19 configurations.

A case study at Google reported a large corpus of builds and build errors mainly focusing on static compilation problems [46]. Beller *et al.* [6] performed an analysis of builds with Travis CI on top of Github. One interesting insight is that about 10% of builds show different behavior when different environments are used. It is related to our endeavor to build the "good" distributed environment for testing JHipster at scale.

9 Conclusion and Future Work

In this article, we reported on the first ever endeavor to test all configurations of an industrial-strength, open-source generator: JHipster. We described the lessons learned and assessed the cost of engineering a configuration-aware testing infrastructure capable of processing 26,000+ configurations in 4,376 hours CPU time. Despite the involved costs, the effort was worth it both in terms of test automation insights and sampling techniques assessment. We found that almost 36% of the configurations fail, caused by six interactions faults. As a result, sampling strategies that cover feature interactions (e.g., dissimilarity or t-wise) are the most effective to find faults. Our recommendations were crossed with JHipster's lead developers opinions which can drive future sampling practices. Based on multiple sources of evidence, we have investigated some of the previous established findings (e.g., effectiveness of t-wise sampling) and also reported novel insights (e.g., for testing at scale). Without the effort of testing all configurations, we would have missed important lessons learned or superficially assess existing techniques.

Our empirical study thus opens opportunities for future work, both for practitioners and researchers. Future work will cover fitting the test budget in continuous integration setting and devise new statistical selection/prioritisation sampling techniques. We plan to continue our collaboration with the JHipster community. Contributors involvement, testing infrastructure evolution and data science challenges (e.g., [30]) are on the agenda. Our long term objective is to pro-

vide evidenced-based theories and tools for continuously testing configurable systems.

Acknowledgements This research was partially funded by the EU Project STAMP ICT-16-10 No.731529 and the Dutch 4TU project "Big Software on the Run".

References

1. Abal, I., Brabrand, C., Wasowski, A.: 42 variability bugs in the linux kernel: A qualitative analysis. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, pp. 421–432. ACM, New York, NY, USA (2014). DOI 10.1145/2642937.2642990. URL http://doi.acm.org/10.1145/2642937.2642990
2. Acher, M., Collet, P., Lahire, P., France, R.B.: FAMILIAR: A domain-specific language for large scale management of feature models. Science of Computer Programming (SCP) **78**(6), 657–681 (2013)
3. Al-Hajjaji, M., Krieter, S., Thüm, T., Lochau, M., Saake, G.: IncLing: efficient product-line testing using incremental pairwise sampling. In: GPCE '16, pp. 144–155. ACM (2016)
4. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines. Springer (2013)
5. Balouek, D., Carpen-amarie, A., Charrier, G., Jeannot, E., Jeanvoine, E., Adrien, L., Niclausse, N., Nussbaum, L.: Adding Virtualization Capabilities to Grid ' 5000 p. 18 (2012)
6. Beller, M., Gousios, G., Zaidman, A.: Oops, my tests broke the build: An explorative analysis of travis ci with github. In: Proceedings of the 14th International Conference on Mining Software Repositories, MSR '17, pp. 356–367. IEEE Press, Piscataway, NJ, USA (2017). DOI 10.1109/MSR.2017.62. URL https://doi.org/10.1109/MSR.2017.62
7. Benavides, D., Segura, S., Ruiz-Corts, A.: Automated analysis of feature models 20 years later: a literature review. Information Systems **35**(6) (2010). DOI 10.1016/j.is.2010.01.001. URL http://dx.doi.org/10.1016/j.is.2010.01.001
8. Classen, A., Boucher, Q., Heymans, P.: A text-based approach to feature modelling: Syntax and semantics of TVL. Science of Computer Programming **76**(12), 1130–1143 (2011). DOI 10.1016/j.scico.2010.10.005
9. Classen, A., Cordy, M., Schobbens, P.Y., Heymans, P., Legay, A., Raskin, J.F.: Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. IEEE Transactions on Software Engineering **39**(8), 1069–1089 (2013)
10. Cohen, M., Dwyer, M., Jiangfan Shi: Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. IEEE Transactions on Software Engineering **34**(5), 633–650 (2008)
11. da Mota Silveira Neto, P.A., do Carmo Machado, I., McGregor, J.D., de Almeida, E.S., de Lemos Meira, S.R.: A systematic mapping study of software product lines testing. Information and Software Technology **53**(5), 407–423 (2011)
12. Engström, E., Runeson, P.: Software product line testing - A systematic mapping study. Information and Software Technology **53**(1), 2–13 (2011)
13. Ganesan, D., Knodel, J., Kolb, R., Haury, U., Meier, G.: Comparing costs and benefits of different test strategies for a software product line: A study from testo ag. In: Software Product Line Conference, 2007. SPLC 2007. 11th International, pp. 74–83. IEEE (2007)
14. Garvin, B.J., Cohen, M.B., Dwyer, M.B.: Evaluating improvements to a meta-heuristic search for constrained interaction testing. Empirical Software Engineering **16**(1), 61–102 (2011)
15. Greiler, M., van Deursen, A., Storey, M.A.: Test confessions: A study of testing practices for plug-in systems. In: ICSE '12, pp. 244–254. ACM (2012)
16. Hahsler, M., Grün, B., Hornik, K.: arules – A computational environment for mining association rules and frequent item sets. Journal of Statistical Software **14**(15), 1–25 (2005). DOI 10.18637/jss.v014.i15. URL http://dx.doi.org/10.18637/jss.v014.i15
17. Halin, A., Nuttinck, A., Acher, M., Devroey, X., Perrouin, G., Heymans, P.: Yo variability! jhipster: A playground for web-apps analyses. In: Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems, VAMOS '17, pp. 44–51. ACM, New York, NY, USA (2017). DOI 10.1145/3023956.3023963. URL http://doi.acm.org/10.1145/3023956.3023963
18. Henard, C., Papadakis, M., Harman, M., Traon, Y.L.: Combining multi-objective search and constraint solving for configuring large software product lines. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1, pp. 517–528 (2015). DOI 10.1109/ICSE.2015.69
19. Henard, C., Papadakis, M., Perrouin, G., Klein, J., Heymans, P., Le Traon, Y.: Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines. IEEE Transactions on Software Engineering **40**(7), 650–670 (2014)
20. Henard, C., Papadakis, M., Perrouin, G., Klein, J., Traon, Y.L.: Pledge: A product line editor and test generation tool. In: Proceedings of the 17th International Software Product Line Conference Co-located Workshops, SPLC '13 Workshops, pp. 126–129. ACM, New York, NY, USA (2013). DOI 10.1145/2499777.2499778. URL http://doi.acm.org/10.1145/2499777.2499778
21. Hervieu, A., Baudry, B., Gotlieb, A.: PACOGEN: Automatic Generation of Pairwise Test Configurations from Feature Models. In: ISSRE '11, i, pp. 120–129. IEEE (2011)
22. JHipsterTeam: Jhipster website (2017). URL https://jhipster.github.io
23. Jin, D., Qu, X., Cohen, M.B., Robinson, B.: Configurations everywhere: implications for testing and debugging in practice. In: ICSE '14 Companion Proceedings, pp. 215–224. ACM (2014)
24. Johansen, M.F.: Pairwiser (2016). URL https://inductive.no/pairwiser/
25. Johansen, M.F., Haugen, Ø., Fleurey, F.: An algorithm for generating t-wise covering arrays from large feature models. In: SPLC '12, vol. 1, p. 46. ACM (2012)
26. Johansen, M.F., Haugen, Ø., Fleurey, F., Eldegard, A.G., Syversen, T.: Generating better partial covering arrays by modeling weights on sub-product lines. In: R.B. France, J. Kazmeier, R. Breu, C. Atkinson (eds.) MoDELS, *Lecture Notes in Computer Science*, vol. 7590, pp. 269–284. Springer (2012)
27. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility

Study. Tech. rep., Carnegie-Mellon University, Software Engineering Institute (1990)
28. Kim, C.H.P., Batory, D.S., Khurshid, S.: Reducing combinatorics in testing product lines. In: Proceedings of the tenth international conference on Aspect-oriented software development, pp. 57–68. ACM (2011)
29. Kim, C.H.P., Marinov, D., Khurshid, S., Batory, D., Souto, S., Barros, P., d'Amorim, M.: Splat: lightweight dynamic analysis for reducing combinatorics in testing configurable systems. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 257–267. ACM (2013)
30. Kim, M., Zimmermann, T., DeLine, R., Begel, A.: The emerging role of data scientists on software development teams. In: Proceedings of the 38th International Conference on Software Engineering, ICSE '16, pp. 96–107. ACM, New York, NY, USA (2016). DOI 10.1145/2884781.2884783. URL http://doi.acm.org/10.1145/2884781.2884783
31. Mathur, A.P.: Foundations of software testing. Pearson Education, India (2008)
32. Medeiros, F., Kästner, C., Ribeiro, M., Gheyi, R., Apel, S.: A comparison of 10 sampling algorithms for configurable systems. In: Proceedings of the 38th International Conference on Software Engineering (ICSE). ACM Press, New York, NY (2016)
33. Meinicke, J., Wong, C.P., Kästner, C., Thüm, T., Saake, G.: On essential configuration complexity: Measuring interactions in highly-configurable systems. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE). ACM Press, New York, NY (2016)
34. Nguyen, H.V., Kästner, C., Nguyen, T.N.: Exploring variability-aware execution for testing plugin-based web applications. In: ICSE '14, pp. 907–918. ACM (2014)
35. Ochoa, L., Pereira, J.A., González-Rojas, O., Castro, H., Saake, G.: A survey on scalability and performance concerns in extended product lines configuration. In: Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems, VAMOS '17, pp. 5–12. ACM, New York, NY, USA (2017). DOI 10.1145/3023956.3023959. URL http://doi.acm.org/10.1145/3023956.3023959
36. Oster, S., Zorcic, I., Markert, F., Lochau, M.: Mosopolite: Tool support for pairwise and model-based software product line testing. In: Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems, VaMoS '11, pp. 79–82. ACM, New York, NY, USA (2011). DOI 10.1145/1944892.1944901. URL http://doi.acm.org/10.1145/1944892.1944901
37. Parejo, J.A., Sánchez, A.B., Segura, S., Ruiz-Cortés, A., Lopez-Herrejon, R.E., Egyed, A.: Multi-objective test case prioritization in highly configurable systems: A case study. Journal of Systems and Software **122**, 287–310 (2016)
38. Perrouin, G., Oster, S., Sen, S., Klein, J., Baudry, B., le Traon, Y.: Pairwise testing for software product lines: Comparison of two approaches. Software Quality Journal **20**(3-4), 605–643 (2011)
39. Perrouin, G., Sen, S., Klein, J., Baudry, B., Traon, Y.l.: Automated and scalable t-wise test case generation strategies for software product lines. In: Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation, ICST '10, pp. 459–468. IEEE Computer Society, Washington, DC, USA (2010). DOI 10.1109/ICST.2010.43. URL http://dx.doi.org/10.1109/ICST.2010.43
40. Qu, X., Cohen, M.B., Rothermel, G.: Configuration-aware regression testing: an empirical study of sampling and prioritization. In: Proceedings of the 2008 international symposium on Software testing and analysis, pp. 75–86. ACM (2008)
41. Raible, M.: The JHipster mini-book. C4Media (2015)
42. Sánchez, A.B., Segura, S., Parejo, J.A., Ruiz-Cortés, A.: Variability testing in the wild: the drupal case study. Software & Systems Modeling **16**(1), 173–194 (2017). DOI 10.1007/s10270-015-0459-z. URL http://dx.doi.org/10.1007/s10270-015-0459-z
43. Sánchez, A.B., Segura, S., Ruiz-Cortés, A.: The drupal framework: A case study to evaluate variability testing techniques. In: Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS '14, pp. 11:1–11:8. ACM, New York, NY, USA (2013). DOI 10.1145/2556624.2556638. URL http://doi.acm.org/10.1145/2556624.2556638
44. Sanchez, A.B., Segura, S., Ruiz-Cortes, A.: A Comparison of Test Case Prioritization Criteria for Software Product Lines. In: ICST '14, pp. 41–50. IEEE (2014)
45. Sayyad, A.S., Menzies, T., Ammar, H.: On the value of user preferences in search-based software engineering: A case study in software product lines. In: ICSE '13, pp. 492–501. IEEE (2013)
46. Seo, H., Sadowski, C., Elbaum, S., Aftandilian, E., Bowdidge, R.: Programmers' build errors: A case study (at google). In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pp. 724–734. ACM, New York, NY, USA (2014). DOI 10.1145/2568225.2568255. URL http://doi.acm.org/10.1145/2568225.2568255
47. Society, I.C., Bourque, P., Fairley, R.E.: Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0, 3rd edn. IEEE Computer Society Press, Los Alamitos, CA, USA (2014)
48. Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G.: A classification and survey of analysis strategies for software product lines. ACM Comput. Surv. **47**(1), 6:1–6:45 (2014). DOI 10.1145/2580950. URL http://doi.acm.org/10.1145/2580950
49. Yilmaz, C., Cohen, M.B., Porter, A.A.: Covering arrays for efficient fault characterization in complex configuration spaces. IEEE Transactions on Software Engineering **32**(1), 20–34 (2006)