CrossMark

Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack

Axel Halin[1] · Alexandre Nuttinck[2] · Mathieu Acher[3] ·
Xavier Devroey[4] (iD) · Gilles Perrouin[5] · Benoit Baudry[6]

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 a lot to be learned by systematically and exhaustively testing a configurable system. In this case study, we report on the first ever endeavour to test

---

Communicated by: Sven Apel

✉ Xavier Devroey
x.d.m.devroey@tudelft.nl

Alexandre Nuttinck
alexandre.nuttinck@cetic.be

Mathieu Acher
mathieu.acher@irisa.fr

Gilles Perrouin
gilles.perrouin@unamur.be

Benoit Baudry
baudry@kth.se

[1] PReCISE, NaDI, Faculty of Computer Science, University of Namur, Namur, Belgium

[2] CETIC, Charleroi, Belgium

[3] Inria, CNRS, IRISA, University of Rennes, Rennes, France

[4] SERG, Delft University of Technology, Delft, The Netherlands

[5] (FNRS research associate) PReCISE, NaDI, Faculty of Computer Science, University of Namur, Namur, Belgium

[6] KTH Royal Institute of Technology, Stockholm, Sweden

all possible configurations of the 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 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.

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 and 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. Ensuring quality for all configurations is a difficult task. For example, Melo et al. compiled 42,000+ random Linux kernels and found that only 226 did not yield any compilation warning (Melo et al. 2016). Though formal methods and program analysis can identify some classes of defects (Thüm et al. 2014; Classen et al. 2013) – leading to variability-aware testing approaches (e.g., Nguyen et al. 2014; Kim et al. 2011, 2013) – a common practice is still to execute and *test* a sample of (representative) configurations. Indeed, enumerating all configurations is perceived as impossible, impractical or both. While this is generally true, we believe there is a lot to be learned by rigorously and exhaustively testing a configurable system. Prior empirical investigations (e.g., Medeiros et al. 2016; Sanchez et al. 2014, 2017) 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 a 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 Medeiros et al. 2016; Meinicke et al. 2016; Hervieu et al. 2011; Henard et al. 2014; Cohen et al. 2008; Sanchez et al. 2014) 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 26,000+ 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 (Halin et al. 2017) 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 compares them with JHipster developers' current practice. In short, we report on the first ever endeavour to test all possible configurations of the industry-strength open-source configurable software system: JHipster. While there have been efforts in this direction for Linux kernels, their variability space forces to focus on subsets (the selection of 42,000+ kernels corresponds to one month of computation Melo et al. 2016) or to investigate bugs qualitatively (Abal et al. 2014, 2018). 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, 4376 hours (∼182 days) CPU time and 5.2 terabytes of available disk space 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 while requiring small samples of configurations. Studying both fault and failure efficiencies provides a more nuanced perspective on the compared techniques;
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 took 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 provides background information on sampling techniques and motivates the case; Section 3 presents the JHipster case study, the research questions, and methodology applied in this article; Section 4 presents the human and computational cost of testing all JHipster configurations; Section 5 presents the faults and failures found during JHipster testing; Section 6 makes a ground truth comparison of the sampling strategies; Section 7 positions our approach with respect to studies comparing sampling strategies on other configurable systems; Section 8 gives the practitioners point of view on JHipster testing by presenting the results of our interview
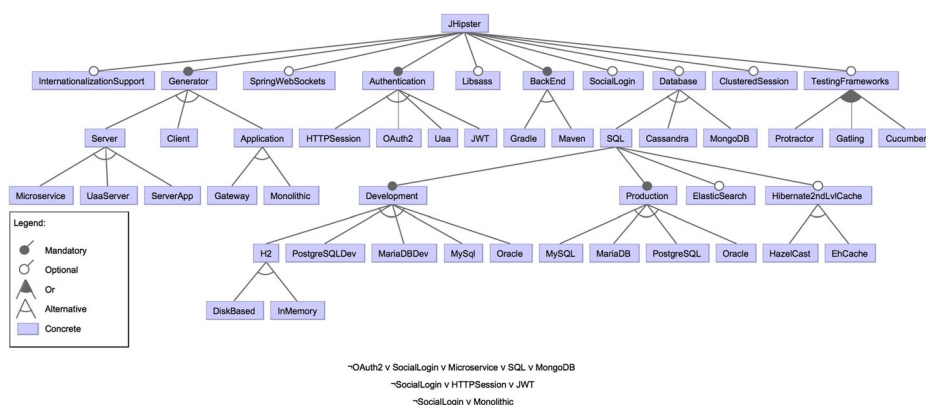
Fig. 1 JHipster reverse engineered feature model (only an excerpt of cross-tree constraints is given)

with JHipster developers; Section 9 discusses the threats to validity; and Section 10 wraps up with conclusions.

2 Background and related work

Configurable systems have long been studied by the Software Product Line (SPL) engineering community (Pohl et al. 2005; Apel et al. 2013a). They use a tree-like structure, called *feature model* (Kang et al. 1990), to represent the set of valid combinations of options: i.e., the variants (also called products). Each option (or *features*[1]) maybe decomposed into sub-features and additional constraints may be specified amongst the different features.

For instance, Fig. 1 presents the full feature model of JHipster. Each JHipster variant has a Generator option that may be either a Server, a Client, or an Application; may also have a Database that is SQL or Cassandra or MongoDB; etc. Additional constraints specify for instance that SocialLogin may only be selected for Monolithic applications.

2.1 Reverse engineering variability models

The first step required to reason on an existing configurable system is to identify its variability. There are some approaches in the literature that attempt to extract variability and synthesize a feature model. For example, She et al. devised a technique to transform the description language of the Linux kernel into a representative feature model (She et al. 2011). The inference of parent-child relationships amongst features proved to be problematic as the well as the mapping of multi-valued options to boolean features. As a result, feature models extracted with such a technique have to be further validated and corrected (Henard et al. 2013a). Abbasi et al. (2014) designed an extraction approach that first look for variability patterns in web configurator tools and complete extracted information using a web crawler. In this case, the feature model is not synthesised. Indeed, static analysis has been largely used to reason about configuration options at the code level (e.g., Rabkin

---

[1]In the remaining of this paper, we consider *features* as units of variability: i.e., *options*.

and Katz 2011; Nadi et al. 2015). Such techniques often lie at the core of variability-aware testing approaches discussed below. As we will detail in our study, the configurator implementation as well as variation points of JHipster are scattered in different kinds of artefacts, challenging the use of static and dynamic analyses. As a result, we rather used a manual approach to extract a variability model. Though automated variability extraction can be interesting to study JHipster evolution over the long term, we leave it out of the scope of the present study.

2.2 Testing a configurable system

Over the years, various approaches have been developed to test configurable systems (da Mota Silveira Neto et al. 2011; Engström and Runeson 2011; Machado et al. 2014). They can be classified into two strategies: *configurations sampling* and *variability-aware testing*. Configuration sampling approaches sample a representative subset of all the valid configurations of the system and test them individually. Variability-aware testing approaches instrument the testing environment to take variability information and reduce the test execution effort.

2.2.1 Variability-aware testing

To avoid re-execution of variants that have exactly the same execution paths for a test case, Kim et al. and Shi et al. use static and dynamic execution analysis to collect variability information from the different code artefacts and remove relevant configurations accordingly (Kim et al. 2013; Shi et al. 2012).

Variability-aware execution approaches (Kim et al. 2011; Nguyen et al. 2014; Austin and Flanagan 2012) instrument an interpreter of the underlying programming language to execute the tests only once on all the variants of a configurable system. For instance, Nguyen et al. implemented Varex, a variability-aware PHP interpreter, to test WordPress by running code common to several variants only once (Nguyen et al. 2014). Alternatively, instead of executing the code, Reisner et al. use a symbolic execution framework to evaluate how the configuration options impact the coverage of the system for a given test suite (Reisner et al. 2010). Static analysis and notably type-checking has been used to look for bugs in configurable software (Kastner and Apel 2008; Kenner et al. 2010). A key point of type-checking approaches is that they have been scaled to very large code bases such as the Linux kernel.

Although we believe that JHipster is an interesting candidate case study for those approaches, with the extra difficulty that variability information is scattered amongst different artefacts written in different languages (as we will see in Section 4.1), they require a (sometimes heavy) instrumentation of the testing environment. Therefore, we leave variability-aware testing approaches outside the scope of this case study and focus instead on configuration sampling techniques that can fit into the existing continuous integration environment of JHipster developers (see Section 8.1).

2.2.2 Configurations sampling

Random sampling This strategy is straightforward: select a random subset of the valid configurations. Arcuri and Briand (2012) demonstrate that, in the absence of constraints between the options, this sampling strategy may outperform other sampling strategies. In our evaluation, random sampling serves as basis for comparison with other strategies.

T-wise sampling T-wise sampling comes from Combinatorial Interaction Testing (CIT), which relies on the hypothesis that most faults are caused by undesired interactions of a small number of features (Kuhn et al. 2004). This technique has been adapted to variability-intensive systems for more than 10 years (Cohen et al. 2008; Lopez-Herrejon et al. 2015).

A t-wise algorithm samples a set of configurations such that all possible *t*-uples of options are represented at least once (it is generally not possible to have each *t*-uples represented exactly once due to constraints between options). Parameter *t* is called *interaction strength*. The most common t-wise sampling is pairwise (2-wise) (Yilmaz et al. 2006; Cohen et al. 2008; Perrouin et al. 2011; Johansen 2016; Hervieu et al. 2011). In our evaluation, we rely on SPLCAT (Johansen et al. 2012a), an efficient t-wise sampling tool for configurable systems based on a greedy algorithm.

Dissimilarity sampling Despite advances being made, introducing constraints during t-wise sampling yields scalability issues for large feature models and higher interaction strengths (Medeiros et al. 2016). To overcome those limitations, Henard et al. developed a dissimilarity-driven sampling (Henard et al. 2014). This technique approximates t-wise coverage by generating dissimilar configurations (in terms of shared options amongst these configurations). From a set of random configurations of a specified cardinality, a (1+1) evolutionary algorithm evolves this set such that the distances amongst configurations are maximal, by replacing a configuration at each iteration, within a certain amount of time. In our evaluation, we rely on Henard et al.'s implementation: PLEDGE (Henard et al. 2013c). The relevance of dissimilarity-driven sampling for software product lines has been empirically demonstrated for large feature models and higher strengths (Henard et al. 2014). This relevance was also independently confirmed for smaller SPLs (Al-Hajjaji et al. 2016).

Incremental Sampling Incremental sampling consists of focusing on one configuration and progressively adding new ones that are related to focus on specific parts of the configuration space (Uzuncaova et al. 2010; Oster et al. 2010; Lochau et al. 2012b). For example, Lochau et al. (2012b) proposed a model-based approach that shifts from one product to another by applying "deltas" to statemachine models. These deltas enable automatic reuse/adaptation of test model and derivation of retest obligations. Oster et al. extend combinatorial interaction testing with the possibility to specify a predefined set of products in the configuration suite to be tested (Oster et al. 2010). Incremental techniques naturally raise the issue of which configuration to start from. Our goal was to compare techniques that explore the configuration space in the large and therefore we did not include incremental techniques in our experiments.

One-disabled sampling The core idea of one-disabled sampling is to extract configurations in which all options are activated but one (Abal et al. 2014; Medeiros et al. 2016). For instance, in the feature diagram of Fig. 1, we will have a configuration where the SocialLogin option is deactivated and all the other options (that are not mandatory) are activated.

This criterion allows various strategies regarding its implementation: in our example, one may select a configuration with a Server xor Client xor Application option active. All those three configurations fit for the one-disabled definition. In their implementation, Medeiros et al. (2016) consider the first valid configuration returned by the solver.

Since SAT solvers rely on internal orders to process solutions (see Henard et al. 2014) 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.

In our evaluation (see Section 6.2), for each disabled option, we choose to apply a random selection of the configuration to consider. Additionally, we also extend this sampling criteria to all valid configurations where one feature is disabled and the others are enabled (called *all-one-disabled* in our results): in our example, for the SocialLogin option deactivated, we will have one configuration with Server option activated, one configuration with Client option activated, and one configuration with Application option activated.

One-enabled sampling This sampling mirrors one-disabled and consists of enabling each option one at a time (Abal et al. 2014; Medeiros et al. 2016). For instance, a configuration where the SocialLogin option is selected and all the other options are deselected. As for one-disabled, for each selected option, we apply a random selection of the configuration to consider in our evaluation; and the criteria are extended to *all-one-enabled*, with all the valid configurations for each selected option.

Most-enabled-disabled sampling This method only samples two configurations: one where as many options as possible are selected and one where as many options as possible are deselected (Abal et al. 2014; Medeiros et al. 2016). If more than one valid configuration is possible for most-enabled (respectively most-disabled) options, we randomly select one most-enabled (respectively most-disabled) configuration. The criteria are extended to *all-most-enabled-disabled*, with all the valid configurations with most-enabled (respectively most-disabled) options.

Other samplings Over the years, many other sampling techniques have been developed. Some of them use other artefacts in combination with the feature model to perform the selection. Johansen et al. (2012b) extended SPLCAT by adding weights on sub-product lines. Lochau et al. combine coverage of the feature model with test model coverage, such as control and data flow coverage (Lochau et al. 2012a). Devroey et al. switched the focus from variability to behaviour (Devroey et al. 2014; Devroey et al. 2016) and usage of the system (Devroey et al. 2017) by considering a featured transition system for behaviour and configurations sampling.

In this case study, we only consider the feature model as input for our samplings and focus on random, t-wise, dissimilarity, one-enabled, one-disabled, and most-enabled-disabled techniques.

2.3 Comparison of sampling approaches

Perrouin et al. (2011) 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. (2011) also used models from the SPLOT repository to produce a small number of configurations. Johansen et al. (2012b) extension of SPLCAT has been applied to the Eclipse IDE and to TOMRA, an industrial product line. Empirical investigations were pursued on larger models (1000 features and above) notably on OS kernels (e.g., Henard et al. 2014; Johansen et al. 2012a) demonstrating the relevance of metaheuristics for large sampling tasks (Henard et al. 2015; Ochoa et al. 2017). However, these comparisons were performed at the model level using artificial faults.

Several authors considered sampling on actual systems. Oster et al. (2011) 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 a higher interaction strength was required (3-wise).

Finally, in Section 7, we present an in-depth discussion of related case studies with sampling techniques comparison.

2.4 Motivation of this Study

Despite the number of empirical investigations (e.g., Ganesan et al. 2007; Qu et al. 2008) and surveys (e.g., Engström et al. 2011; Thüm et al. 2014; Da Mota Silveira Neto et al. 2011) 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 (Meinicke et al. 2016; Yilmaz et al. 2006). And provide a complete, open, and reusable dataset to the configurable system testing community to evaluate and compare new approaches.

3 Case study

JHipster is an open-source, industrially used generator for developing Web applications (JHipsterTeam: JHipster website 2017). 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

```
(...)
@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(...);
   }
   <%_ } _%>
(...)
```

Listing 1 Variability in _DatabaseConfiguration.java

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 Yeoman[2] (aka *yo*) tool to scaffold the application (Raible 2015). JHipster relies on conditional compilation with EJS[3] 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 variants; (ii) there are 48 configuration options and a configurator guides user throughout different questions. Not all combinations of options are possible and there are 15 constraints between options; (ii) variability is scattered among numerous kinds of artefacts (*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 (Jin et al. 2014).

This complexity challenges core developers and contributors of JHipster. Unsurprisingly, numerous configuration faults have been reported on mailing lists and eventually fixed with commits.[4] Though formal methods and variability-aware program analysis can identify some defects (Thüm et al. 2014; Classen et al. 2013; Nguyen et al. 2014), 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 (Medeiros et al. 2016; Meinicke et al. 2016; Hervieu et al. 2011; Henard et al. 2014; Cohen et al. 2008; Sanchez et al. 2014). Due to the reasonable number of options and the presence of 15 constraints, we (as researchers) also have a unique opportunity to gather a ground truth through the testing of *all* configurations.

3.1 Research questions

Our research questions are formulated around three axes: the first one addresses the feasibility of testing all JHipster configurations; the second question addresses the bug-discovery power of state-of-the-art configuration samplings; and the last one addresses confronts our results with the JHipster developers point of view.

3.1.1 (RQ1) What is the feasibility of testing all JHipster configurations?

This research question explores the cost of an exhaustive and automated testing strategy. It is further decomposed into two questions:

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

---

[2]http://yeoman.io/

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

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

To answer this first question, we reverse engineered a feature model of JHipster based on various code artefacts (described in Section 4.1), and devise an analysis workflow to automatically derive, build, and test JHipster configurations (described in Section 4.2). This workflow has been used to answer our second research question:

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

To keep a manageable execution time, the workflow has been executed on the INRIA Grid'5000, a large-scale testbed offering a large amount of computational resources (Balouek et al. 2012).

Section 4.4 describes our engineering efforts in building a fully automated testing infrastructure for all JHipster variants. We also evaluate the computational cost of such an exhaustive testing; describe the necessary resources (man-power, time, machines); and report on encountered difficulties as well as lessons learned.

3.1.2 (RQ2) To what extent can sampling help to discover defects in JHipster?

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" (Mathur 2008; Society et al. 2014) (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 5), a single fault can explain many configuration failures since the same feature interactions cause the failure.

To compare different sampling approaches, the first step is to characterise failures and faults that can be found in JHipster:

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

Based on the outputs of our analysis workflow, we identify the faults causing one or more failures using statistical analysis (see Section 5.2) and confirm those faults using qualitative analysis, based on issue reports of the JHipster GitHub project (see Section 5.3).

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?

We consider the sampling techniques presented in Section 2.2.2; all techniques use the feature model as primary artefact (see Section 6) to perform the sampling. For each sampling technique, we measure the failures and the associated faults that the sampled configurations detect. Besides a comparison between automated sampling techniques, we also compare the manual sampling strategy of the JHipster project.

Since our comparison is performed using specific results of JHipster's executions and cannot be generalized as such, we confront our findings to other case studies found in the literature. In short:

(RQ2.3)   How do our sampling techniques effectiveness findings compare to other case studies and works?

To answer this question, we perform a literature review on empirical evaluation of sampling techniques (see Section 7).

3.1.3 *(RQ3) How can sampling help JHipster developers?*

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

(RQ3.1)   What is the most cost-effective sampling strategy for JHipster?

And confront our findings to the current development practices of the JHipster developers:

(RQ3.2)   What are the recommendations for the JHipster project?

To answer this question, we performed a semi-structured interview of the lead developer of the project and exchanged e-mails with other core developers to gain insights on the JHipster development process and collect their reactions to our recommendations, based on an early draft of this paper (see Section 8).

3.2 Methodology

We address these questions through quantitative and qualitative research. We initiated the work in September 2016 and selected JHipster 3.6.1[5] (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 for four 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.

4 All configurations testing costs (RQ1)

4.1 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 tries of every possible combinations has scalability issues. Some answers activate or deactivate some questions and options. As a result, we rather

---

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

considered the source code from GitHub for identifying options and constraints. Though options are scattered amongst artefacts, there is a central place that manages the configurator and then calls different sub-generators to derive a variant.

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., Kang et al. 1990) to benefit from state-of-the-art reasoning techniques developed in software product line engineering (Benavides et al. 2010; Classen et al. 2011; Apel et al. 2013a; Thüm et al. 2014; Acher et al. 2013). 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 modelling 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;
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 Fig. 1: 48 identified features and 15 constraints (we only present four of them in Fig. 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. However, we decided to adapt the initial model as follows:

1. we added Docker as a new optional feature (Docker) to denote the fact that the deployment may be performed using Docker or using Maven or Gradle. Docker has been

```
(...)
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
(...)
```

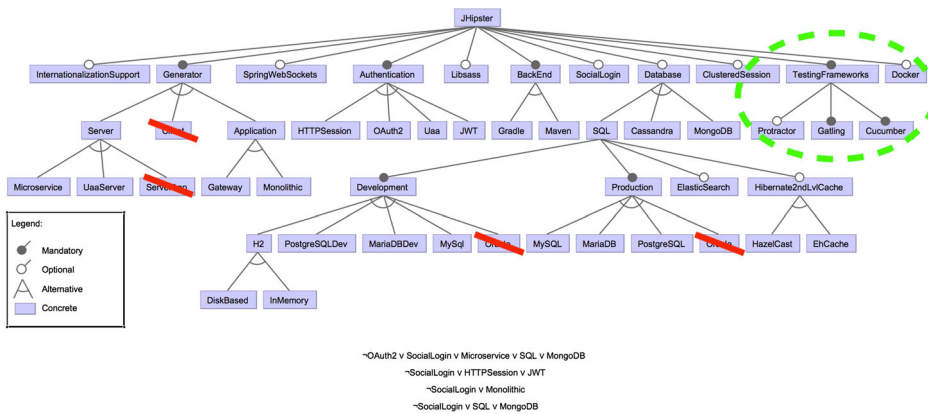**Listing 2** Configurator: server/prompt.js (excerpt)

Fig. 2 JHipster specialised feature model used to generate JHipster variants (only an excerpt of cross-tree constraints is given)

introduced in JHipster 3.0.0 and is present by default in all generated variants (and therefore does not appear in the feature model of Fig. 1). However, when running JHipster, the user may choose to use it or not, hence the definition of Docker as optional for our analysis workflow: when the option is selected, the analysis workflow performs the deployment using 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 Section 4.2).

Strictly speaking, we test *all* configurations of a *specialized* JHipster, presented in Fig. 2. This specialization can be thought of a test model, which focusses on the most relevant open source variants. 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 Fig. 1 augmented with specialized constraints that negate features Oracle12c, Oracle, ServerApp, and Client (in red in Fig. 2) and that add an optional Docker feature and make Gatling and Cucumber features mandatory (in green in Fig. 2). This specialization leads to a total of **26,256** variants.

4.2 Fully automated derivation and testing

From the feature model, we enumerated all valid configurations using solvers and FAMIL-IAR (Acher et al. 2013). We developed a comprehensive workflow for testing each configuration. Figure 3 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.

Fig. 3 Testing workflow of JHipster configurations

4.2.1 Engineering a configurable system for testing configurations
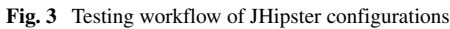
The execution of a unique and generic command for testing JHipster variants was not directly possible. For instance, the build of a JHipster application relies either on Maven or Gradle, two alternative features of our variability model. We developed variability-aware scripts to execute commands specific 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 configurable system; (ii) the development was iterative and mainly consisted in *automating all tasks* originally considered as manual (e.g., starting database services).

4.2.2 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

```
@Test
public void testAuthenticatedUser() (...) {
    restUserMockMvc.perform(get("/api/authenticate")
        .with(request -> {
            request.setRemoteUser("test");
            return request;})
        .accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(content().string("test"));
}
```

Listing 3 JHipster generated JUnit test in `AccountResourceIntTest.java`

and tested. Technically, we need to populate Web applications with *entities* (i.e., structured data like tables in an 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.[6] 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 (*Entities Generation* in Fig. 3), we run several tests: integration tests written in Java using the *Spring Test Context* framework (see Listing 3 for instance), user interface tests written in JavaScript using the *Karma.js* framework (see Listing 4 for instance), etc., and create an executable JHipster variant (*Build Maven/Gradle* in Fig. 3). The tests run at this step are automatically generated and include defaults tests common to all JHipster variants and additional tests generated by the JDL entities creation. On average, the Java line coverage is 44.53% and the JavaScript line coverage is 32.19%.

We instantiate the generated entities (*Entities Populating* in Fig. 3) using the Web user interface through Selenium scripts. We integrate the following testing frameworks to compute additional metrics (*Tests* in Fig. 3): Cucumber, Gatling and Protractor. We also implement generic oracles that analyse and extract log error messages. And finally, repeated two last steps using Docker (*Build Docker*, *Entities Populating*, and *Tests* in Fig. 3) before saving the generated log files.

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.

4.2.3 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. In the end, we converged on an all-inclusive environment.

---

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

```
describe('Component Tests', () => {
  describe('LoginComponent', () => {
    (...)
    it('should redirect user when register', () => {
      // WHEN
      comp.register();
      // THEN
      expect(mockActiveModal.dismissSpy).
          toHaveBeenCalledWith('to state register');
      expect(mockRouter.navigateSpy).
          toHaveBeenCalledWith(['/register']);
    });
    (...)
  });
});
```

Listing 4 JHipster generated Karma.js test in `user.service.spec.ts`

4.2.4 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,[7] a large-scale testbed offering a large amount of computational resources (Balouek et al. 2012). 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 pre-set Debian images.

4.2.5 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 481MB and a node_modules (for JavaScript dependencies) of 249MB. Furthermore, we build a Docker variant right after the classical build (see Fig. 3) to derive two JHipster variants (with and without Docker) without restarting the whole derivation process.

4.2.6 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).

---

[7]https://www.grid5000.fr

4.3 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.

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.

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**.

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.

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).

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

4.4 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 2,325 configurations with the same error message: "Communications link failure", denoting network communication error (between a node and the controller for instance) on the grid. Those

failures have been ignored and configurations have been re-run again afterwards to have consistent results.

> **(RQ1.2)** *What are the computational resources needed to test all configurations?* Testing all configurations requires a significant amount of computational resources (4,376 hours CPU time and 5.2 terabytes of disk space).

5 Results of the testing workflow execution (RQ2.1)

The execution of the testing workflow yielded a large file comprising numerous results for each configuration. This file[8] 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.

5.1 Bugs: A quick inventory

Out of the **26,256** configurations we tested, we found that **9,376** (**35.70**%) failed. This failure occurred either during the compilation of the variant (*Compilation* in Fig. 3) or during its packaging as an executable Jar file (*Build Maven/Gradle* in Fig. 3, which includes execution of the different Java and JavaScript tests generated by JHipster), although the generation (*App generation* in Fig. 3) was successful. We also found that some features were more concerned by failures as depicted in Fig. 4. Regarding the application type, for instance, *microservice gateways* and *microservice applications* are 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 (4114 configurations) fail to deploy.

5.2 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 failures' results, we decided to use the Association Rule learning method (Hahsler et al. 2005). It aims at extracting relations (called *rules*) between variables of large data-sets. The Association Rule method is well suited to find the (combinations of) features leading to a failure, out of tested configurations.

Formally and adapting the terminology of association rules, the problem can be defined as follows.

– let $F = \{ft_1, ft_2, \ldots, ft_n, bs\}$ be a set of n features (ft_i) plus the status of the build (bs), i.e., build failed or not;
– let $C = \{c_1, c_2, \ldots, c_m\}$ be a set of m configurations.

---

[8]Complete results are available at https://github.com/xdevroey/jhipster-dataset/tree/master/v3.6.1.
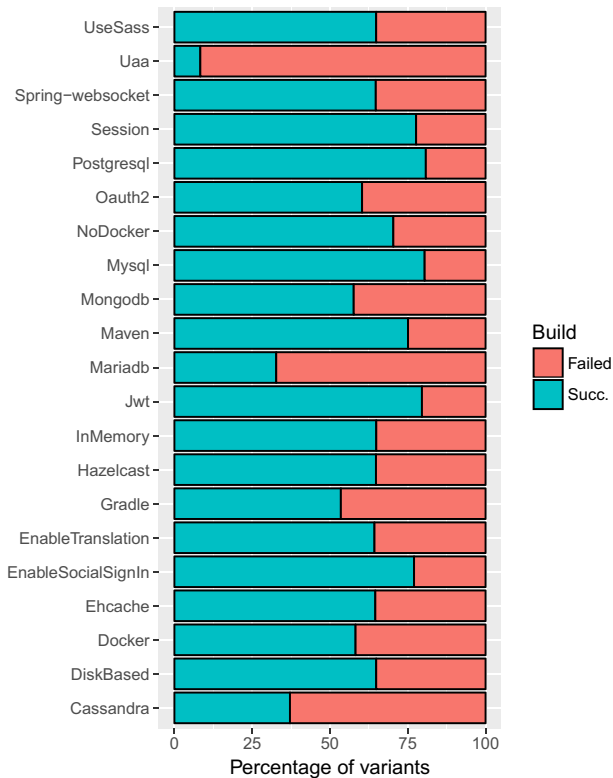
Fig. 4 Proportion of build failure by feature

Each configuration in *C* has a unique identifier and contains a subset of the features in *F* and the status of its build. A rule is defined as an implication of the form: $X \Rightarrow Y$, where $X, Y \subseteq F$.

The outputs of the method are a set of rules, each constituted by:

- *X* the *left-hand side (LHS)* or antecedent of the rule;
- *Y* the *right-hand side (RHS)* or consequent of the rule.

For our problem, we consider that *Y* is a single target: the status of the build. For example, we want to understand what combinations of features lead to a failure, either during the compilation or the build process. To illustrate the method, let us take a small example (see Table 1).

The set of features is $F = \{$ *mariadb*, *gradle*, *enableSocialSignIn*, *websocket*, *failure* $\}$ and in the table is shown a small database containing the configurations, where, in each entry, the value 1 means the presence of the feature in the corresponding configuration, and the value 0 represents the absence of a feature in that configuration. In Fig. 1, when *build failure* has the value 1 (resp. 0), it means the build failed (resp. succeeded). An example rule could be:

$$\{\text{mariadb, graddle}\} \Rightarrow \{\text{build failure}\}$$

Meaning that if *mariadb* and *gradle* are activated, configurations will not build.

As there are many possible rules, some well-known measures are typically used to select the most interesting ones. In particular, we are interested in the *support*, the proportion

Table 1 An example of JHipster data (feature values and build status for each configuration)

Conf.	gradle	mariadb	enableSocialSignIn	websocket	...	build failure
1	1	0	0	0	...	0
2	0	1	0	0	...	0
3	0	0	1	1	...	0
4	1	1	0	0	...	1
5	1	0	0	0	...	0
6	1	1	0	0	...	1
...	...	...	...	...	...	...

We want to extract association rules stating which combinations of feature values lead to a build failure (e.g., gradle)

of configurations where LHS holds and the *confidence*, the proportion of configurations where both LHS and RHS hold. In our example and for the rule {mariadb, graddle} ⇒ {build failure}, the support is 2/6 while the confidence is 1.

Table 2 gives some examples of the rules we have been able to extract. We parametrized the method as follows. First, we restrained ourselves to rules where the RHS was a failure: either *Build=KO* (build failed) or *Compile=KO* (compilation failed). 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.[9] 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 2). We found no rule involving 1 or 3 features. We decided to examine 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 2 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 5 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 99.1% of the failures.

5.3 Qualitative analysis

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

---

[9]https://cran.r-project.org/web/packages/arules/

Table 2 Association rules involving compilation and build failures

| Id	Left-hand side	Right-hand side	Support	Conf.	GitHub Issue	Report/Correction date
MoSo	DatabaseType="mongodb", EnableSocialSignIn=true	Compile=KO	0.488%	1	4037	27 Aug 2016 (report and fix for milestone 3.7.0)
MAGr	prodDatabaseType="mariadb", buildTool="gradle"	Build=KO	16.179%	1	4222	27 Sep 2016 (report and fix for milestone 3.9.0)
UADo	Docker=true, authentication-Type="uaa"	Build=KO	6.825%	1	UAA is in Beta	Not corrected
OASQL	authenticationType="uaa", hibernateCache="no"	Build=KO	2.438%	1	4225	28 Sep 2016 (report and fix for milestone 3.9.0)
UAEH	authenticationType="uaa", hibernateCache="ehcache"	Build=KO	2.194%	1	4225	28 Sep 2016 (report and fix for milestone 3.9.0)
MADo	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)
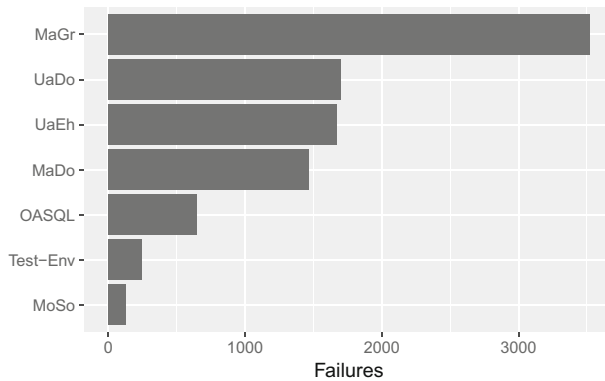
Fig. 5 Proportion of failures by fault described in Table 2

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 **1468** 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.

MariaDB using Gradle This second fault concerns variants relying on Gradle as build tool and MariaDB as the database (**3519** configurations). It is caused by a missing dependency in template file *server/template/gradle/_liquibase.gradle*.

UAA authentication with Docker The third fault occurs in Microservice Gateways or Microservice applications using an UAA server as authentication mechanism (**1703** 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.

UAA authentication with Ehcache as Hibernate second 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 **1667** configurations.

OAuth2 authentication with SQL database This defect is faced **649** times, when trying to deploy a web-app, using an 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.

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.

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.[10] 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).

Feature interaction strength Our findings show that only two features are involved in five (out of six) faults, and four features are involved in the last fault. The prominence of 2-wise interactions is also found in other studies. Abal et al. report that, for the Linux bugs they have qualitatively examined, more than a half (22/43) are attributed to 2-wise interactions (Abal et al. 2018). Yet, for different interaction strengths, there is no common trend: we do not have 3-wise interactions while this is second most common case in Linux, we did not find any fault caused by one feature only.

> **(RQ2.1)** *How many and what kinds of failures/faults can be found in all configurations?* Exhaustive testing shows that almost 36% of the configurations fail. Our analysis identifies 6 interaction faults as the root cause for this high percentage.

6 Sampling techniques comparison (RQ2.2)

In this section, we first discuss the sampling strategy used by the JHipster team. We then use our dataset to make a ground truth comparison of six state-of-the-art sampling techniques.

6.1 JHipster team sampling strategy

The JHipster team uses a sample of 12 representative configurations for the version 3.6.1, to test their generator (see Section 8.1 for further explanations on how these were sampled). During a period of several weeks, the testing configurations have been used at *each commit* (see also Section 8.1). 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 any defect.

6.2 Comparison of sampling techniques

As testing all configurations is very costly (see **RQ1**), sampling techniques remain of interest. We would like to find as many failures and faults as possible 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).

---

[10]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 address **RQ2.2** with numerous sampling techniques considered in the literature (Perrouin et al. 2010b; Johansen et al. 2012a; Abal et al. 2014; Medeiros et al. 2016). For each technique, we report on the number of failures and faults.

6.2.1 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* (Johansen et al. 2012a), which has the advantage of being deterministic: for one given feature model, it will always provide the same sample.

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. It has to be noted that the discovery of a 4-wise interaction fault with a 3-wise setting is a 'collateral' effect (Petke et al. 2013), since any sample covering completely t-way interactions also yields an incomplete coverage of higher-order interactions.

One-disabled sampling Using **one-disabled** sampling algorithm, we extract configurations in which all features are activated but one.

To overcome any bias in selecting the first valid configuration, as suggested by Medeiros et al. (2016), we applied a random selection instead. We therefore select a valid random configuration for each disabled feature (called **one-disabled** in our results) and repeat experiments 1,000 times 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-one-disabled** configurations (i.e., all valid configurations where one feature is disabled and the other are enabled). 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 (Abal et al. 2014; Medeiros et al. 2016).

As for one-disabled, we choose to randomly select valid configurations instead of taking the first one returned by the solver. Repeating the experiment 1,000 times: 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 all the **6 faults**. **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 (Henard et al. 2014; Al-Hajjaji et al. 2016) using **PLEDGE** (Henard et al. 2013c).

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 (Henard et al. 2013c). 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 2,500. 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.

6.2.2 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*. For both metrics, a high efficiency is desirable since it denotes a small sample with either a high failure or fault detection capability.

The results are summarized in Table 3. We present in Fig. 6a (respectively, Fig. 6b) 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 Medeiros et al. 2016) 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.

Table 3 Efficiency of different sampling techniques (bold values denote the highest efficiencies)

Sampling technique	Sample size	Failures (σ)	Failures eff.	Faults (σ)	Fault eff.
1-wise	8	2.000 (N.A.)	25.00%	2.000 (N.A.)	25.00%
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%
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%
4-wise	374	161.000 (N.A.)	**43.05%**	6.000 (N.A.)	1.60%
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%
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%
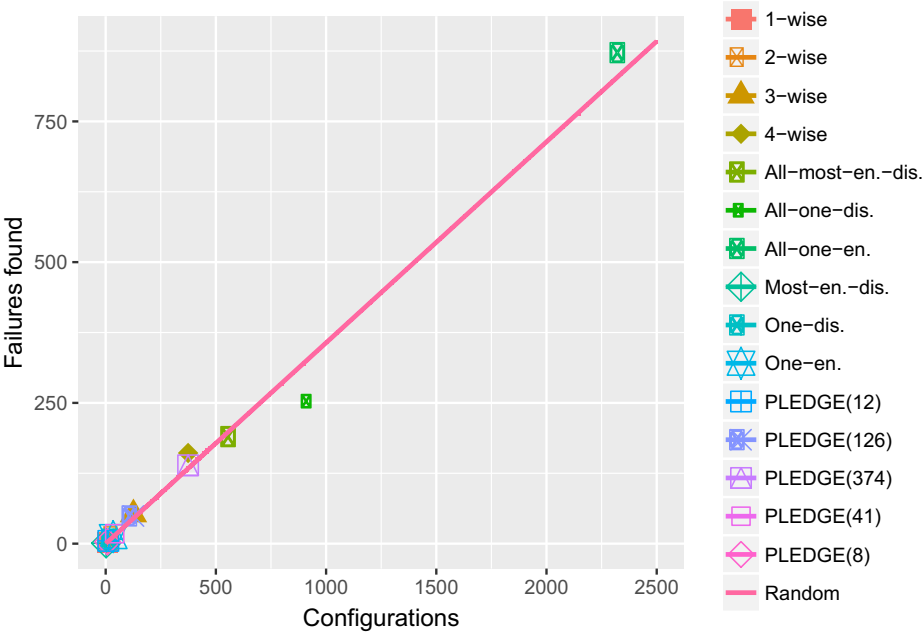
(a) Failures found by sampling techniques
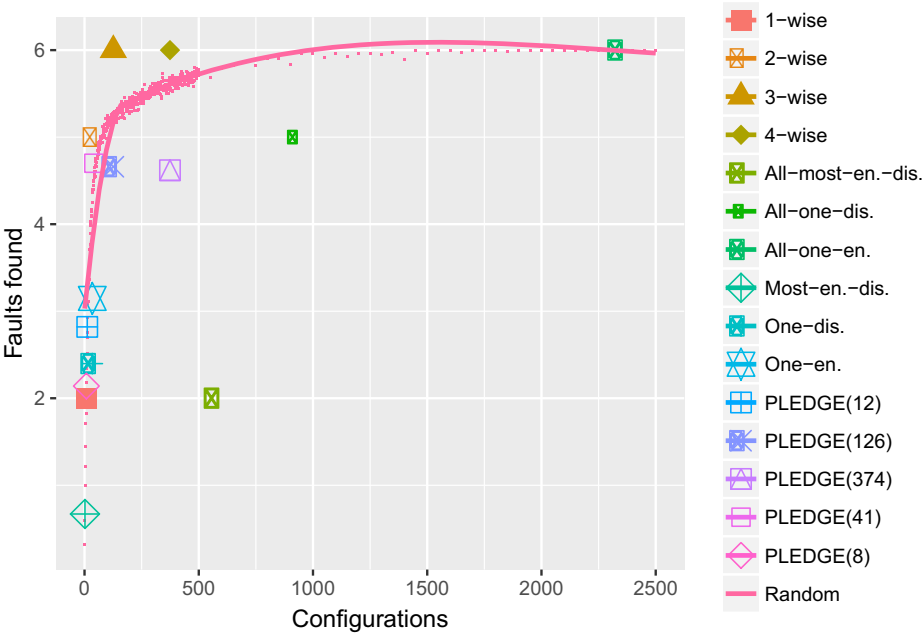
(b) Faults found by sampling techniques

Fig. 6 Defects found by sampling techniques

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 techniques 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.

6.2.3 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. For other cases, finding the faulty features or feature interactions can be much more tricky. In such contexts, investigating *many failures* and using statistical methods (such as association rules) can be helpful to determine the faulty features and their undesired interactions. As a result, the ability of finding failures may be more important than in JHipster case. A trade-off between failure and fault efficiency can certainly be considered when choosing the sampling technique.

> **(RQ2.2)** *How effective are sampling techniques comparatively?* To summarise: (i) random is a strong baseline for failures and faults; (ii) 2-wise and 3-wise sampling are slightly more efficient to find faults than random; (iii) most-enabled-disabled is efficient on average to detect faults but requires to be lucky; (iv) dissimilarity is superior to random for small sample sizes; (v) a small sample is sufficient to identify most important faults, there is no need to cover the whole configuration space; and (vi) there is no correlation between failure and fault efficiencies.

7 Comparison with Other Studies (RQ2.3)

This section presents a literature review of case studies of configuration sampling approaches to test variability intensive systems.

Specifically, we aim to compare our findings with state-of-the-art results: *Are sampling techniques as effective in other case studies? Do our results confirm or contradict findings*

in other settings? This question is important for (1) practitioners in charge of establishing a suitable strategy for testing their systems; (ii) researchers interested in building evidence-based theories or tools for testing configurable systems.

We first present our selection protocol of relevant papers and an overview of the selected ones. We then confront and discuss our findings from Section 6.2 w.r.t. those studies.

7.1 Studies selection protocol

We consider the following criteria to select existing studies: (i) (ii) configuration sampling approaches are evaluated regarding defects detection capabilities; (iii) evaluation has been performed on an industrial size (open source or not) system (i.e., we discard toy examples); and (iv) evaluation has been performed on the system (possibility to analyse the source code and/or to run the variants to reproduce bugs and failures). We thus discard evaluations that are solely based on feature models such as Perrouin et al. (2011).

We looked for studies in previous literature reviews on product line testing (Engström and Runeson 2011; da Mota Silveira Neto et al. 2011; Machado et al. 2014). They are a common way to give an overview of a research field: e.g., they organise studies according to topic(s) and validation level (for instance, from Machado et al. (2014): no evidence, toy example, opinions or observations, academic studies, industrial studies, or industrial practices).

Before 2014 (i.e., before the publication of the systematic literature review from Machado et al. 2014), empirical evaluations of configurations sampling approaches are focused on their capability to select a sampling satisfying t-wise criteria in a reasonable amount of time or with fewer configurations Ensan et al. 2012; Henard et al. 2013b; Hervieu et al. 2011; Johansen et al. 2011, 2012a; Kim et al. 2013; Lochau et al. 2012a; Perrouin et al. 2010a; Perrouin et al. 2011. We discarded them as they do not match our selection criteria.

To select relevant studies without performing a full systematic literature survey, we applied a forward and backward snowballing search (Jalali and Wohlin 2012). Snowballing is particularity relevant in our case, given the diversity of terms used in the research literature (product line, configurable systems, etc.) and our goal to compare more than one sampling approach. Searching and filtering studies from literature databases would require a large amount of work with few guarantees on the quality of the result. We started the search with two empirical studies known by the authors of the paper: Medeiros et al. (2016) and Sánchez et al. (2017). Those studies are from two different research sub-communities on variability-intensive system testing, configurable systems research (Medeiros et al. Medeiros et al. 2016) and software product line research (Sánchez et al. 2017), which mitigates the risk of missing studies of interest. Eventually, we collected 5 studies presented in Table 4 and we discuss them below.

7.2 Selected studies

Medeiros et al. (2016) compared 10 sampling algorithms using a a corpus of 135 known configuration-related faults from 24 open-source C systems. Like for JHipster, the systems use conditional compilation (#ifdef) to implement variability.

Sánchez et al. (2017) studied Drupal,[11] a PHP web content management, to assess test case prioritization, based on functional and non-functional data extracted from Drupal's Git

---

[11] https://www.drupal.org

Table 4 Evaluation of configuration sampling techniques in the literature

Reference	Samplings	Validation
Medeiros et al. (2016)	Statement-coverage, one-enabled,one-disabled, most-enabled-disabled, random, pair-wise, five-wise, six-wise	135 configuration-related faults in 24 open-source C (`#ifdef`) configurable systems three-wise, four-wise,
Sánchez et al. (2017)	Pairwise	Drupal (PHP modules based Web content management system)
Parejo et al. (2016)	Multi-objective	Drupal (PHP modules based Web content management system)
Souto et al. (2017)	random, one-enabled, one-disabled, most-enabled enabled disabled and pairwise computed from SPLat (Kim et al. 2013)	8 small SPLs + GCC (50 most used options). Samplings' sizes and number of failures were considered.
Apel et al. (2013b)	one-wise, pairwise, three-wise compared to a family-based stategy and enumeration of all products	3 configurable systems written in C and 3 in JAVA

repository. Sánchez et al. assimilate a Drupal module to a feature and performed extensive analysis of Drupal's Git repository and issue tracking system to identify faults and other functional and non-functional attributes (e.g., feature size, feature cyclomatic complexity, number of test for each feature, feature popularity, etc.).

Sánchez et al. consider 2-wise to sample the initial set of configurations to prioritize.

Parejo et al. (2016) extend (Sánchez et al. 2017) work by defining multi-objectives test case selection and prioritization. The algorithm starts with a set of configurations samples, each satisfying 2-wise coverage, and evolves them in order to produce one prioritized configuration sample that maximize the defined criteria. Since new configurations may be injected during the evolution, the algorithm does not only prioritize (unlike for Sánchez et al. 2017), but modifies the samples. Objectives are functional (e.g., dissimilarity amongst configurations, pairwise coverage, cyclomatic complexity, variability coverage, etc.) and non-functional (e.g., number of changes in the features, feature size, number of faults in previous version, etc.).

Souto et al. (2017) explore the tension between soundness and efficiency for configurable systems. They do so by extending SPLat (Kim et al. 2013), a variability-aware dynamic technique that, for each test, monitors configuration variables accessed during test execution and change their values to run the test on new configurations, stopping either when no new configurations can be dynamically explored or a certain threshold is met. The extension, called S-SPLAT (Sampling with SPLat) uses dynamically explored configurations in the goal of meeting a particular sampling criterion. Six heuristics are considered: random, one-enabled, one-disabled, most-enabled-disabled and 2-wise in addition of the original SPLat technique. Experiments carried out on eight SPLs and the GCC compiler considered efficiency (number of configurations explored per test) and efficacy (number of failures detected).

Apel et al. investigated the relevance of family-based model checking (thanks to the SPLVerifier tool chain developed by the authors) with respect to t-wise sampling strategies and analysis all products independently on C and JAVA configurable systems (Apel et al. 2013b). The metrics used were the analysis time and the number of faults founds statically. While both the family-based strategy and the product-based (all-products) strategies covered all the faults (by construction), there is a clear advantage in favour of the former with respect to execution time. Sampling strategies were in between these two extremes in which 3-wise appears to the best compromise.

7.3 Comparison of findings

7.3.1 Sampling effectiveness

Souto et al. reported that one-disabled and a combination involving one-enabled, one-disabled, most-enabled-disabled as well as pairwise appeared to be good compromises for detecting failures. Regarding GCC, one-enabled and most-enabled-disabled were the most appropriate choices. On the one hand, our findings concur with their results:

– 2-wise is indeed one of the most effective sampling technique, capable of identifying numerous failures and the 5 most important faults;
– most-enabled-disabled is also efficient to detect failures (34.15%) and faults (33.5% on average).

On the other hand, our results also show some differences:

– one-enabled and one-disabled perform very poorly in our case, requiring a substantial number of configurations to find either failures or faults;
– despite a high fault efficiency, most-enabled-disabled is only able to capture 0.670 faults on average, thus missing important faults.

Medeiros et al. 's results show that most-enabled-disabled offers a good compromise for faults finding ability. On the one hand, our findings concur with their results – most-enabled-disabled is indeed efficient to detect faults (33.5% on average). On the other hand, our experiments reveal an important insight. Amongst valid configurations matching the most-enabled-disabled criterion, some may not reveal any fault. It is the case in our study: the first configurations returned by the solver (as done in Medeiros et al. 2016) discovered 0 fault. For a fair comparison, we thus caution researchers to use a random strategy instead of picking the first configurations when assessing most-enabled-disabled. Besides Medeiros et al. reported that 2-wise is an efficient sampling technique. We concur with this result.

Putting all together our findings and results of Souto et al. and Medeiros et al. , we can recommend the following: *most-enabled-disabled is an interesting candidate to initiate the testing of configurations; 2-wise can be used to complement and continue the effort in order to find further faults.*

Sánchez et al. have chosen 2-wise (using the ICPL algorithm (Johansen et al. 2012a) and CASA Garvin et al. 2011) to sample the initial set of configurations of Drupal. Their results suggest that 2-wise is an efficient sampling technique (though we ignore how pairwise competes with other sampling strategies). As a follow up of their work on Drupal, Parejo et al. concluded that a combination of 2-wise and other objectives (e.g., based on non-functional properties) is usually effective.

In our case, 2-wise is more efficient to identify faults than random, offering a good balance between sample size and fault detection. Overall *our findings on 2-wise concur with the results of Sánchez et al. and Parejo et al.*

Apel et al. considered various sampling strategies. 3-wise appears to be the best compromise, offering a good balance between execution time and fault-detection ability. In our case, 3-wise is slightly more efficient to identify faults than random and can identify the 6 most important faults. However, the important size of 3-wise sampling (126 configurations) degrades its fault efficiency. In particular, 2-wise offers a better trade-off between sampling size and fault detection – it only misses one fault despite having divided the number of configurations to assess by three (41).

Overall, we concur with the findings of Apel et al. There is no need to consider all configurations and *t-wise samplings offer a good trade-off between sampling size and fault detection*. The value of *t* (e.g., 2-wise or 3-wise) and the underlying trade-off should then be debated along the specificities of the software engineering context – it is the role of the next section in which we gather qualitative insights from JHipster community.

7.3.2 Failure vs fault

In our case study, we have made an explicit effort to compute and differentiate failures from faults. We have shown there is no correlation between failure efficiency and fault efficiency. Some of the prior works tend to consider either failures or faults, but very rarely both. There are actually very good reasons for that. On the one hand, the identification of failures requires to execute the configurations in real settings – the process can be very costly and hard to engineer even for a scientific experiment. On the other hand, some works succeed to produce and collect many failures but ignore the actual correspondences with faults.

Though we understand the underlying reasons and difficulties, our point is that the assessment of sampling techniques may highly vary depending on the metric considered (failure or fault efficiency). For example, all-one-enabled has a failure efficiency of 37.26% but is one of the worst techniques in terms of fault rate due of its high sample size. Our recommendation for researchers is to properly report and investigate the distinction (if any) between failures and faults. It is actually an open research direction to further characterize this distinction in other software engineering contexts than JHipster.

7.3.3 Fault corpus

For the assessment of sampling techniques, one need is to define a correspondence between configurations and faults. As reported in the literature and in this study, this task is not trivial, time-consuming, and error-prone. A typical attitude is to manually build a corpus of faults with results confirmed by the developers, or from issues reported in mailing list or bug tracking systems. For example, Sánchez et al. performed extensive analysis of Drupal's Git repository and issue tracking system to identify faults. A possible and important threat is that the corpus of faults is *incomplete*. It can bias the empirical results since some faults may not be considered in the study.

In our case study, we had a unique opportunity to collect *all* faults through the testing of all configurations. Meanwhile we have been able to check whether these faults have been reported by JHipster developers. Our findings show that 6 important faults have been reported (see Table 2). Though some of the faults were missing and required a manual

investigation, they only impact a few configurations comparatively to faults reported on GitHub issues.

Overall, our findings suggest that a corpus of faults coming from an issue tracking system is a good approximation of the real corpus of faults. It is a positive result for other studies based on a manually collected corpus.

> **(RQ2.3)** *How do our findings w.r.t. sampling effectiveness compare to other studies?*
> (i) **From a practical point of view**: We concur with previous findings that show that most-enabled-disabled is an interesting candidate to initiate the testing of configurations. For identifying further faults (and possibly all), we confirm that 2-wise or 3-wise provides a good balance between sampling size and fault-detection capability. (ii) **From a researcher point of view**: Our results show that the assessment of sampling techniques may highly vary depending on the metrics used (failure or fault efficiency). Besides, a corpus of faults coming from an issue tracking system (GitHub) is a good approximation of the real, exhaustive corpus of faults. It is reassuring for research works based on a manually collected corpus.

8 Practitioners viewpoint (RQ3)

We interviewed the JHipster lead developer, Julien Dubois, for 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 developed, used, and tested. We also aimed to confront our empirical results with their current practice.

8.1 JHipster's testing strategy

8.1.1 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.

8.1.2 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 buggier than those with Maven – we learned that Gradle is used in less than 20% of the 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.

> **(RQ3.1)** *What is the most cost-effective sampling strategy for JHipster?* Exhaustive testing 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.

8.2 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[12] 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.

8.3 Discussions

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

8.3.1 Sampling strategy

Our empirical results suggest using 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 trade-off between cost, popularity, and effectiveness of sampling techniques. We see this as an opportunity to further experiment with multi-objective techniques (Sayyad et al. 2013; Parejo et al. 2016; Henard et al. 2015).

8.3.2 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*

---

[12]https://github.com/jhipster/jhipster-devbox

tripling the number of combinations [...] We settled on 19 configurations to keep build times within acceptable limits".[13]

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.

8.3.3 Configuration-aware testing infrastructure

In any case, we recommend developing 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.

> **(RQ3.2)** *What are the recommendations for the JHipster project?* To summarise, recommendations (and challenges) are: (i) for a budget of 19 configurations, dissimilarity is the most appropriate sampling strategy; (ii) the trade-off between cost, popularity, and effectiveness suggests to further experiment with multi-objective techniques; (iii) crowdsourcing the testing effort would help to face the computational cost of testing JHipster; (iv) the development and maintenance of a configuration-aware testing infrastructure is mandatory to automate JHipster testing.

9 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. We also used the Java and JavaScript tests generated by JHipster,

---

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

as well as the data from only one entity model template (the one provided by the JHipster team). 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. Despite those limitations, we found all faults reported by the JHipster community and even new failures.

For the other remaining 242 configurations that fail due to our test infrastructure (see Section 5.3), 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 Beller et al. 2017). 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.

To mitigate the threat related to missed studies comparing findings of configuration sampling techniques, we used a snowballing approach. We started from mapping studies and systematic literature reviews known by the authors. Selected studies have been reviewed by at least three authors.

10 Conclusion and perspectives

In this article, we reported on the first ever endeavour 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.

10.1 Synthesis of lessons learned

Infrastructure costs Building a configuration-aware testing infrastructure for JHipster requires a **substantial effort both in terms of human and computational resources**: **8 man-months** for building the infrastructure and **4376 hours of CPU time** as well as **5.2 terabytes** of disk space used to build and run JHipster configurations. The most difficult part of realising the infrastructure was **to validate it**, especially in a distributed setting. These costs are system-dependent: for example, the Linux project provides tools to compile distinct random kernels, which can be used for various analyses (e.g., Melo et al. 2016; Henard et al. 2013d), and ease the realisation of a testing infrastructure.

Comparing sampling techniques Almost **36%** of the 26,0000 configurations fail. More than **99%** of these failures can be attributed to **six interaction faults up to 4-wise**. The remaining failures are false positives. As a result, in our case, t-wise testing techniques provide guarantees to **find all the faults**. Nevertheless, such guarantees come at a **price**, i.e., the number of configurations to sample (126). Still, only a **small subset** of the total number of configurations is necessary, validating the relevance of sampling techniques. Dissimilarity sampling is slightly better at finding failures though offering **generally good efficiencies w.r.t t-wise with a flexible budget**. Most-enabled-disabled can be **very efficient regarding the very small number of configurations** it requires but should incorporate **randomness** in the sampling algorithm to not rely on a SAT solver's internal order (Henard et al. 2014). Indeed, random sampling remains **a strong baseline** for failure and faults. Finally, investigation of **both** faults and failures efficiencies shows that **they are not correlated** and that it is difficult to optimise them for a single sampling technique. Without the effort of testing

all configurations, we would have missed important lessons or have superficially assessed existing techniques.

Comparison with other studies Our assessment of sampling techniques on JHipster confirm findings of the literature: **most-enabled-disabled is a relevant technique** in order to initiate testing, while t-wise techniques with low values of t provide interesting fault and failure detection ratios.

However, ranking sampling techniques **is highly sensitive to the metrics considered**, which **complicates** the "*comparison of comparing studies*". Yet, fault corpora issued from issue tracking systems such as GitHub **seem to contain almost all issues**, adding relevance to fault mining efforts.

Comparison with JHipster team testing practice Confronting our results to the core JHipster developers was extremely valuable. First, the testing budget is **key**: even for 26,000+ configurations the developers did not have the possibility to test **more than twelve of them** continuously. Additionally, their strategy based on popular configurations **did not find any fault**. Such a stringent budget is a challenge for sampling techniques and combinatorial interaction testing ones in particular. In this case, dissimilarity is **our recommendation**. **Cost, popularity and fault-finding abilities** appeared as important factors in the determination of samples. This pushes for experimenting with multi-objective techniques in such contexts. Finally, our effort in providing **an automated configuration-aware testing infrastructure is mandatory** for viable JHipster testing.

10.2 Test them all, is it worth it?

Our investigations allow us to answer to the key question we raised in the title of this article: *Is it worth testing all configurations?* Obviously, there is no universal 'yes' or 'no' answer to this question as it typically depends on the audience to which the question is asked.

From a **researcher's perspective**, the answer is definitely 'yes'. This effort enabled us to obtain a *ground truth* notably on the faults for this specific version of JHipster. Sampling techniques can then be compared with respect to an absolute value (**all** the faults), which is a stronger evidence than a comparison on a configuration subset. Building and running configurations also gave insights on failures that are not frequently analysed. This enthusiasm should be tempered with respect to the high costs of building a testing infrastructure capable of handling all configurations. Finally, JHipster has the interesting properties of being widely used as an open-source stack, non-trivial but still manageable in terms of configurations, which made this study possible. For example, researchers working on Linux kernels cannot envision to answer this question in the current state of computing since Linux has thousands of options (Nadi et al. 2015; Abal et al. 2014; Melo et al. 2016).

From a **practitioner's perspective**, the answer is a resounding 'no'. The JHipster community cannot afford computing and human costs involved with such an initiative. In the improbable event of having the sufficient resources, validation time at each release or commit would still be a no-go, point that is also likely to hold in other cases. Moreover, we have shown that sampling 126 configurations (out of 26,000+) is enough to find all the faults. While the absolute ranking between sampling methods is variable amongst studies and cases analysed, sampling techniques are more efficient at faults and failures than exhaustive testing, as illustrated by the poor 0.02% of fault efficiency when sampling all configurations. Though testing all configurations is not worth it, we recommend to develop a testing infrastructure *able* to handle all possible configurations; it is a mandatory prerequisite before instrumenting a cost-effective sampling strategy.

10.3 Perspectives

Our empirical study 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, including how techniques like association rules work in such settings. We plan to continue our collaboration with the JHipster community. Contributors involvement, testing infrastructure evolution and data science challenges (e.g., Kim et al. 2016) are on the agenda. Our long-term objective is to provide evidence-based theories and tools for continuously testing configurable systems.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

References

Abal I, Brabrand C, Wasowski A (2014) 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. ACM, New York, pp 421–432

Abal I, Melo J, Stanciulescu S, Brabrand C, Ribeiro M, Wasowski A (2018) Variability bugs in highly configurable systems: A, qualitative analysis. ACM Trans Softw Eng Methodol 26(3):10:1–10:34

Abbasi EK, Acher M, Heymans P, Cleve A (2014) Reverse engineering web configurators. In: 2014 software evolution week - IEEE conference on software maintenance, reengineering, and reverse engineering, CSMR-WCRE 2014, Antwerp, pp 264–273

Acher M, Collet P, Lahire P, France RB (2013) FAMILIAR: a domain-specific language for large scale management of feature models. Sci Comput Programm (SCP) 78(6):657–681

Al-Hajjaji M, Krieter S, Thüm T, Lochau M, Saake G (2016) IncLing: efficient product-line testing using incremental pairwise sampling. In: Proceedings of the 2016 ACM SIGPLAN international conference on generative programming: concepts and experiences - GPCE 2016. ACM, pp 144–155

Apel S, Batory D, Kästner C, Saake G (2013a) Feature-oriented software product lines. Springer, Berlin

Apel S, von Rhein A, Wendler P, Größlinger A, Beyer D (2013b) Strategies for product-line verification: case studies and experiments. In: Proceedings of the 2013 international conference on software engineering, ICSE '13. IEEE, Piscataway, pp 482–491

Arcuri A, Briand L (2012) Formal analysis of the probability of interaction fault detection using random testing. IEEE Trans Softw Eng 38(5):1088–1099

Austin TH, Flanagan C (2012) Multiple facets for dynamic information flow. ACM SIGPLAN Not 47(1):165

Balouek D, Carpen-Amarie A, Charrier G, Desprez F, Jeannot E, Jeanvoine E, Lèbre A, Margery D, Niclausse N, Nussbaum L, Richard O, Pérez C, Quesnel F, Rohr C, Sarzyniec L (2012) Adding virtualization capabilities to Grid'5000. Research Report RR-8026, INRIA. https://hal.inria.fr/hal-00720910

Beller M, Gousios G, Zaidman A (2017) 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. IEEE Press, Piscataway, pp 356–367

Benavides D, Segura S, Ruiz-Cortés A (2010) Automated analysis of feature models 20 years later: a literature review. Inf Syst 35(6):615–636

Classen A, Boucher Q, Heymans P (2011) A text-based approach to feature modelling: syntax and semantics of TVL. Sci Comput Program 76(12):1130–1143

Classen A, Cordy M, Schobbens PY, Heymans P, Legay A, Raskin JF (2013) Featured transition systems: foundations for verifying variability-intensive systems and their application to LTL model checking. IEEE Trans Softw Eng 39(8):1069–1089

Cohen M, Dwyer M, Shi J (2008) Constructing interaction test suites for highly-configurable systems in the presence of constraints: a greedy approach. IEEE Trans Softw Eng 34(5):633–650

da Mota Silveira Neto PA, do Carmo Machado I, McGregor JD, de Almeida ES, de Lemos Meira SR (2011) A systematic mapping study of software product lines testing. Inf Softw Technol 53(5):407–423

Devroey X, Perrouin G, Legay A, Cordy M, Schobbens PY, Heymans P (2014) Coverage criteria for behavioural testing of software product lines. In: Margaria T, Steffen B (eds) Leveraging applications of formal methods, verification and validation. Technologies for mastering change: 6th international symposium, ISoLA 2014, proceedings, Part I, LNCS, vol 8802. Springer, Corfu, pp 336–350

Devroey X, Perrouin G, Legay A, Schobbens PY, Heymans P (2016) Search-based similarity-driven behavioural SPL testing. In: Proceedings of the tenth international workshop on variability modelling of software-intensive systems - VaMoS '16. ACM Press, Salvador, pp 89–96

Devroey X, Perrouin G, Cordy M, Samih H, Legay A, Schobbens PY, Heymans P (2017) Statistical prioritization for software product line testing: an experience report. Softw Syst Model 16(1): 153–171

Engström E, Runeson P (2011) Software product line testing - A systematic mapping study. Inf Softw Technol 53(1):2–13

Ensan F, Bagheri E, Gašević D (2012) Evolutionary search-based test generation for software product line feature models. In: Ralyté J, Franch X, Brinkkemper S, Wrycza S (eds) Advanced Information Systems Engineering: 24th International Conference, CAiSE '12. Springer, pp 613–628

Ganesan D, Knodel J, Kolb R, Haury U, Meier G (2007) 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. IEEE, pp 74–83

Garvin BJ, Cohen MB, Dwyer MB (2011) Evaluating improvements to a meta-heuristic search for constrained interaction testing. Empir Softw Eng 16(1):61–102

Hahsler M, Grün B, Hornik K (2005) arules – A computational environment for mining association rules and frequent item sets. J Stat Softw 14(15):1–25

Halin A, Nuttinck A, Acher M, Devroey X, Perrouin G, Heymans P (2017) 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. ACM, New York, pp 44–51

Henard C, Papadakis M, Perrouin G, Klein J, Le Traon Y (2013a) Towards automated testing and fixing of re-engineered feature models. In: Proceedings of the 2013 international conference on software engineering, ICSE '13. IEEE Press, Piscataway, pp 1245–1248

Henard C, Papadakis M, Perrouin G, Klein J, Traon YL (2013b) Multi-objective test generation for software product lines. In: Proceedings of the 17th international software product line conference, SPLC '13. ACM Press, pp 62–71

Henard C, Papadakis M, Perrouin G, Klein J, Traon YL (2013c) 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. ACM, New York, pp 126–129

Henard C, Papadakis M, Perrouin G, Klein J, Traon YL (2013d) Towards automated testing and fixing of re-engineered feature models. In: Notkin D, Cheng BHC, Pohl K (eds) 35th international conference on software engineering, ICSE '13. IEEE Computer Society, San Francisco, pp 1245–1248

Henard C, Papadakis M, Perrouin G, Klein J, Heymans P, Le traon Y (2014) Bypassing the combinatorial explosion: using similarity to generate and prioritize t-wise test configurations for software product lines. IEEE Trans Softw Eng 40(7):650–670

Henard C, Papadakis M, Harman M, Traon YL (2015) Combining multi-objective search and constraint solving for configuring large software product lines. In: 2015 IEEE/ACM 37th IEEE international conference on software engineering - ICSE '15, vol 1, pp 517–528

Hervieu A, Baudry B, Gotlieb A (2011) PACOGEN: Automatic generation of pairwise test configurations from feature models. In: IEEE 22nd international symposium on software reliability engineering - ISSRE '11, i. IEEE, pp 120–129

Jalali S, Wohlin C (2012) Systematic literature studies: database searches vs. backward snowballing. In: Proceedings of the ACM-IEEE international symposium on empirical software engineering and measurement - ESEM '12. ACM, p 29

JHipsterTeam: JHipster website (2017) Https://jhipster.github.io, Accessed Feb 2017

Jin D, Qu X, Cohen MB, Robinson B (2014) Configurations everywhere: implications for testing and debugging in practice. In: Companion Proceedings of the 36th international conference on software engineering - ICSE companion 2014. ACM, pp 215–224

Johansen MF, Haugen Ø, fleurey F (2011) Properties of realistic feature models make combinatorial testing of product lines feasible. In: Whittle J, Clark T, Kůhne T (eds) Model driven engineering languages and systems: 14th international conference, MODELS '11, Section 3. Springer, pp 638–652

Johansen MF, Haugen Ø, Fleurey F (2012a) An algorithm for generating t-wise covering arrays from large feature models. In: Proceedings of the 16th international software product line conference on - SPLC '12 -volume 1, vol. 1. ACM, p 46

Johansen MF, Haugen Ø, Fleurey F, Eldegard AG, Syversen T (2012b) Generating better partial covering arrays by modeling weights on sub-product lines. In: France RB, Kazmeier J, Breu R, Atkinson C (eds) Proceedings of the 15th international conference on model driven engineering languages and systems - MoDELS '12, Lecture notes in computer science, vol 7590. Springer, pp 269–284

Johansen MF (2016) Pairwiser. https://inductive.no/pairwiser/

Kang K, Cohen S, Hess J, Novak W, Peterson AS (1990) Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute

Kastner C, Apel S (2008) Type-checking software product lines-a formal approach. In: 2008 23rd IEEE/ACM international conference on automated software engineering - ASE '08. IEEE, pp 258–267

Kenner A, Kästner C, Haase S, Leich T (2010) Typechef: toward type checking #ifdef variability in c. In: Proceedings of the 2nd international workshop on feature-oriented software development, FOSD '10. ACM, New York, pp 25–32

Kim CHP, Batory D, Khurshid S (2011) Reducing combinatorics in testing product lines. In: Proceedings of the tenth international conference on aspect-oriented software development, AOSD '11. ACM, pp 57–68

Kim CHP, Marinov D, Khurshid S, Batory D, Souto S, Barros P, d'Amorim M (2013) Splat: lightweight dynamic analysis for reducing combinatorics in testing configurable systems - esec/fse '13. In: Proceedings of the 2013 9th joint meeting on foundations of software engineering. ACM, pp 257–267

Kim M, Zimmermann T, DeLine R, Begel A (2016) The emerging role of data scientists on software development teams. In: Proceedings of the 38th international conference on software engineering, ICSE '16. ACM, New York, pp 96–107

Kuhn D, Wallace D, Gallo A (2004) Software fault interactions and implications for software testing. IEEE Trans Softw Eng 30(6):418–421

Lochau M, Oster S, Goltz U, Schůrr A (2012a) Model-based pairwise testing for feature interaction coverage in software product line engineering. Softw Qual J 20(3-4):567–604

Lochau M, Schaefer I, Kamischke J, Lity S (2012b) Incremental model-based testing of delta-oriented software product lines. In: Brucker A, Julliand J (eds) Tests and proofs, LNCS, vol 7305. Springer, pp 67–82

Lopez-Herrejon RE, Fischer S, Ramler R, Egyed A (2015) A first systematic mapping study on combinatorial interaction testing for software product lines. In: 2015 IEEE eighth international conference on software testing, verification and validation workshops (ICSTW). IEEE, pp 1–10

Machado IdC, McGregor JD, Cavalcanti YC, de Almeida ES (2014) On strategies for testing software product lines: a systematic literature review. Inf Softw Technol 56(10):1183–1199

Mathur AP (2008) Foundations of software testing. Pearson Education, India

Medeiros F, Kästner C, Ribeiro M, Gheyi R, Apel S (2016) A comparison of 10 sampling algorithms for configurable systems. In: Proceedings of the 38th international conference on software engineering - ICSE '16. ACM Press, Austin, pp 643–654

Meinicke J, Wong CP, Kästner C, Thüm T, Saake G (2016) On essential configuration complexity: measuring interactions in highly-configurable systems. In: Proceedings of the 31st IEEE/ACM International conference on automated software engineering - ASE 2016, 2. ACM Press, Singapore, pp 483–494

Melo J, Flesborg E, Brabrand C, Wąsowski A (2016) A quantitative analysis of variability warnings in linux. In: Proceedings of the tenth international workshop on variability modelling of software-intensive systems, VaMoS '16. ACM, pp 3–8

Nadi S, Berger T, Kästner C, Czarnecki K (2015) Where do configuration constraints stem from? an extraction approach and an empirical study. IEEE Trans Softw Eng 41(8):820–841

Nguyen HV, Kästner C, Nguyen TN (2014) Exploring variability-aware execution for testing plugin-based web applications. In: Proceedings of the 36th international conference on software engineering - ICSE '14. ACM, pp 907–918

Ochoa L, Pereira JA, González-Rojas O, Castro H, Saake G (2017) 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. ACM, pp 5–12

Oster S, Markert F, Ritter P (2010) Automated incremental pairwise testing of software product lines. In: Software product lines: going beyond. Springer, pp 196–210

Oster S, Zorcic I, Markert F, Lochau M (2011) Moso-polite: 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. ACM, New York, pp 79–82

Parejo JA, Sánchez AB, Segura S, Ruiz-Cortés A, Lopez-Herrejon RE, Egyed A (2016) Multi-objective test case prioritization in highly configurable systems: a case study. J Syst Softw 122:287–310

Perrouin G, Sen S, Klein J, Baudry B, Le Traon Y (2010a) Automated and scalable t-wise test case generation strategies for software product lines. In: 2010 third international conference on software testing, verification and validation, ICST '10. IEEE, pp 459–468

Perrouin G, Sen S, Klein J, Baudry B, Traon Y (2010b) 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. IEEE Computer Society, Washington, pp 459–468

Perrouin G, Oster S, Sen S, Klein J, Baudry B, Le Traon Y (2011) Pairwise testing for software product lines: comparison of two approaches. Softw Qual J 20(3-4):605–643

Petke J, Yoo S, Cohen MB, Harman M (2013) Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In: Proceedings of the 2013 9th joint meeting on foundations of software engineering, ESEC/FSE 2013. ACM, pp 26–36

Pohl K, Böckle G, Van Der linden F (2005) Software product line engineering: foundations, principles, and techniques. Springer, Berlin

Qu X, Cohen MB, Rothermel G (2008) Configuration-aware regression testing: an empirical study of sampling and prioritization. In: Proceedings of the 2008 international symposium on software testing and analysis - ISSTA '08. ACM, pp 75–86

Rabkin A, Katz R (2011) Static extraction of program configuration options. In: Proceedings of the 33rd international conference on software engineering, ICSE '11. ACM, pp 131–140

Raible M (2015) The JHipster mini-book. C4Media

Reisner E, Song C, Ma KK, Foster JS, Porter A (2010) Using symbolic evaluation to understand behavior in configurable software systems. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering, ICSE '10, vol 1. ACM Press, p 445

Sanchez AB, Segura S, Ruiz-Cortes A (2014) A comparison of test case prioritization criteria for software product lines. In: 2014 IEEE seventh international conference on software testing, verification and validation - ICST. IEEE, pp 41–50

Sánchez AB, Segura S, Parejo JA, Ruiz-Cortés A (2017) Variability testing in the wild: the drupal case study. Softw Syst Model 16(1):173–194

Sayyad AS, Menzies T, Ammar H (2013) On the value of user preferences in search-based software engineering: A case study in software product lines. In: ICSE'13. IEEE, pp 492–501

She S, Lotufo R, Berger T, Wasowski A, Czarnecki K (2011) Reverse engineering feature models. In: Proceedings of the 33rd international conference on software engineering, ICSE '11. ACM, New York, pp 461–470

Shi J, Cohen MB, Dwyer MB (2012) Integration testing of software product lines using compositional symbolic execution. In: Proceedings of the 15th international conference on fundamental approaches to software engineering, LNCS, vol 7212. Springer, pp 270–284

Society IC, Bourque P, Fairley RE (2014) Guide to the software engineering body of knowledge (SWE-BOK(r)): version 3.0, 3rd edn. IEEE Computer Society Press, Los Alamitos

Souto S, D'Amorim M, Gheyi R (2017) Balancing soundness and efficiency for practical testing of configurable systems. In: 2017 IEEE/ACM 39th international conference on software engineering (ICSE). IEEE, pp 632–642

Thüm T, Apel S, Kästner C, Schaefer I, Saake G (2014) A classification and survey of analysis strategies for software product lines. ACM Comput Surv 47(1):6:1–6:45

Uzuncaova E, Khurshid S, Batory D (2010) Incremental test generation for software product lines. IEEE Trans Softw Eng 36(3):309–322

Yilmaz C, Cohen MB, Porter AA (2006) Covering arrays for efficient fault characterization in complex configuration spaces. IEEE Trans Softw Eng 32(1):20–34

Axel Halin received his masters degree in Computer Science from the University of Namur, Belgium in 2017. During his final year, he studied configurable systems and studied JHipster as a case study in his master thesis. Since September 2017, he works as a software engineer in the industry.

Alexandre Nuttinck is a research engineer at CETIC, an applied research center in Belgium since September 2017. His main field of activity concerns cloud computing. He received a masters degree in Computer Science, from the University of Namur, Belgium in 2017. In his last year of study, he carried out a research internship at INRIA Rennes where he focused on the automation of testing JHipster configurations.

Dr. Mathieu Acher is Associate Professor at University of Rennes 1 / Inria, France since 2012. His research focuses on reverse engineering, modelling and learning software variability. He has authored papers in top international venues (ESEC/FSE, ASE, SPLC, MoDELS, IJCAI, etc.). He is the scientific leader of VaryVary, a research project about machine learning and configurable software systems.

Dr. Xavier Devroey is a post-doctoral researcher at the Delft University of Technology, where he is involved in the EU Software Testing AMPlification (STAMP) and the 3TU Big Software on the Run (BSR) projects. He received his Ph.D. in Computer Science from the University of Namur. His main research interests are mutation testing, search-based and model-based software testing, test suite augmentation, and variability-intensive systems.

Dr. Gilles Perrouin received the joint PhD degree from the University of Luxembourg and Namur. Since 2017, he is an FNRS research associate (tenured scientist) hosted at the Faculty of Computer Science, University of Namur, Belgium. He is a member of the PReCISE research center and the Namur Digital Institute, in which he explores software product lines, modelling and model-based testing.

Dr. Benoit Baudry is a Professor in Software Technology at the Royal Institute of Technology (KTH) in Stockholm, Sweden. He received his PhD in 2003 from the University of Rennes and was a research scientist at INRIA from 2004 to 2017. His research is in the area of software testing, code analysis and automatic diversification. His work is published in software engineering top journals and conferences, and it is funded by the European Union, the Wallenberg foundation and software companies.