Yo Variability! JHipster: A Playground for Web-Apps Analyses

Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, Patrick Heymans

▶ **To cite this version:**

Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, et al.. Yo Variability! JHipster: A Playground for Web-Apps Analyses. 11th International Workshop on Variability Modelling of Software-intensive Systems, Feb 2017, Eindhoven, Netherlands. pp.44 - 51, 10.1145/3023956.3023963 . hal-01468084

HAL Id: hal-01468084

https://inria.hal.science/hal-01468084

Submitted on 15 Feb 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Yo Variability!
JHipster: A Playground for Web-Apps Analyses

Axel Halin
PReCISE Research Center
University of Namur, Belgium
axel.halin@student.unamur.be

Alexandre Nuttinck
PReCISE Research Center
University of Namur, Belgium
alexandre.nuttinck@student.unamur.be

Mathieu Acher
IRISA, University of Rennes I, France
mathieu.acher@irisa.fr

Xavier Devroey
PReCISE Research Center
University of Namur, Belgium
xavier.devroey@unamur.be

Gilles Perrouin
PReCISE Research Center
University of Namur, Belgium
gilles.perrouin@unamur.be

Patrick Heymans
PReCISE Research Center
University of Namur, Belgium
patrick.heymans@unamur.be

ABSTRACT

Though variability is everywhere, there has always been a shortage of publicly available cases for assessing variability-aware tools and techniques as well as supports for teaching variability-related concepts. Historical software product lines contains industrial secrets their owners do not want to disclose to a wide audience. The open source community contributed to large-scale cases such as Eclipse, Linux kernels, or web-based plugin systems (Drupal, WordPress). To assess accuracy of sampling and prediction approaches (bugs, performance), a case where all products can be enumerated is desirable. As configuration issues do not lie within only one place but are scattered across technologies and assets, a case exposing such diversity is an additional asset. To this end, we present in this paper our efforts in building an explicit product line on top of JHipster, an industrial open-source Web-app configurator that is both manageable in terms of configurations ($\approx$ 163,000) and diverse in terms of technologies used. We present our efforts in building a variability-aware chain on top of JHipster's configurator and lessons learned using it as a teaching case at the University of Rennes. We also sketch the diversity of analyses that can be performed with our infrastructure as well as early issues found using it. Our long term goal is both to support students and researchers studying variability analysis and JHipster developers in the maintenance and evolution of their tools.

CCS Concepts

•**Software and its engineering** → **Software testing and debugging; Empirical software validation;** *Software configuration management and version control systems; Software product lines;* •**Social and professional topics** → **Software engineering education;**

Keywords

Case Study; Web-apps; Variability-related Analyses

1. INTRODUCTION

JHipster [20] is an open-source generator for Web applications (Web-apps). Started in 2013 by Julien Dubois, JHipster aims at supporting all cumbersome aspects of Web applications development: choice of technologies on the client and server sides as well as integrating them in a complete building process. On the server side, JHipster relies on a Java stack (with Spring Boot). On the client side AngularJS and Bootstrap (a HTML/CSS and JavaScript framework) are used. Finally, Yeoman, Bower, Gulp and Maven automate the building process, including the management of dependencies across the offered technologies [20]. JHipster is used all over the world both by independent developers and large companies[1] such as Adobe, Google, HBO, *etc.*.

The setup of a Web-app with JHipster is performed in two phases: *configuration* and *generation*. The configuration is done via a command-line interface (see Figure 1) through which the user can select the technologies that will be included. The result of this configuration is a `yo-rc.json` file (see Listing 1) used for the generation phase. To achieve this generation, JHipster relies on Yeoman[2] using *npm* and *Bower* tools to manage dependencies, and *yo* tool to scaffold projects or useful pieces of an application [41]. Based on the content of the `yo-rc.json` file, JHipster's generator produces relevant artefacts (Java classes and so on). Beyond this generator – the main focus of this paper – JHipster offers multiple sub-generators and even has its own language, *JHipster Domain Language*, to easily generate entities and all related artefacts (e.g., Spring Service Beans).

From its inception to this day, JHipster has constantly grown throughout 146 releases. It now has more than 5000 stars on GitHub and can count on a community of 250 contributors. In October 2016, it has been downloaded 22739 times[3]. This constant evolution allows JHipster to offer up-to-date frameworks and technologies to its users (for instance, the infrastructure can be generated using Docker since release 3.0.0).

By combining configuration and generation in a constantly evolving stack of technologies, JHipster is akin to Mr Jourdain's prose: a software product line initiative without naming it as such. In this paper, we describe our efforts in building an explicit product line on top of JHipster to expose it as a case for research, education and to ease the development of JHipster itself. Our preliminary infrastructure applied on only 300 variants (out of $\approx$ 160,000) already disclosed some unreported issues, which we perceive as an incentive to pur-

[1] https://jhipster.github.io/companies-using-jhipster/
[2] http://yeoman.io/
[3] https://www.npmjs.com/package/generator-jhipster

Listing 1: _yo-rc.json excerpt

```
{
  "generator-jhipster": {
    (...)
    "useSass": false,
    "applicationType": "monolith",
    "testFrameworks": [],
    "jhiPrefix": "jhi",
    "enableTranslation": false
  }
}
```

Figure 1: JHipster command line interface

Listing 2: server/prompt.js excerpt

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

sue in this direction. Though "lifting" such infrastructure in the web domain is not new (*e.g.*, [43, 36]), JHipster offers interesting assets beyond replication studies: *(a)* it covers key aspects of product line development, variability, product derivation and evolution; *(b)* the number of variants is large enough to require automated derivation support (on top of Yeoman) but small enough to be enumerated through distributed computing facilities yielding exact results to assess various kinds of analyses; *(c)* it allows to address variability modelling and configuration challenges across technological spaces [21]. All sources of our preliminary infrastructure can be found at https://github.com/axel-halin/Thesis-JHipster.

In the remainder, we present our efforts to manually reverse engineer variability from JHipster artefacts and define a Web-app software product line (Section 2). We analyse current state of the art for products' analyses, family-based analyses, and product line evolution in Section 3, and presents the JHipster's potential for each of those research fields. Section 4 reports our experiences in using JHipster as an education case study for software product line teaching. Finally, Section 5 concludes this paper and presents future works using JHipster.

2. JHIPSTER AS A PRODUCT LINE

Although never explicitly acknowledged by the JHipster developer, it is straightforward to think JHipster supported technologies[4] (microservice architecture, authentication, *etc.*) as variation points to be resolved during product line application engineering.

Based on this vision, we decided to model the system in a feature model using FAMILIAR[1]. This decision was motivated by our will to assess automatically a maximum of configurations authorized by the JHipster generators. The first step was to identify the variability. To do so, we retrieved the publicly available source code[5] and analysed it. We quickly identified interesting artefacts: `prompts.js` files. JHipster's Yeoman generator is divided in multiple `prompts.js` files, each of which handles specific parts of the configuration process. For instance, `client/prompts.js` offers the possibility to use *LibSass*, while, as illustrated in Listing 2, the type of database is selected in `server/prompts.js`.

From these artefacts, we derived the feature model presented in Figure 2. In this model, the abstract features represent the multiple choices questions (typically, which of these technologies do you wish to use?) while the concrete features are the different choice(s) available to the user. Except for the testing frameworks, all of these multiple-choice questions are exclusive (choose only one production database, for instance), mapped as alternate groups. Yes or no questions are represented by optional features. So, if we consider Listing 2 as an example we have: *database* as an optional abstract feature, with *SQL*, *Mongodb* and *Cassandra* as concrete alternate sub-features. We also identified several constraints in the JavaScript files (`when (...) return applicationType === 'microservice'`, in Listing 2, is one of them) which we synthesized in 15 constraints. For the sake of conciseness, we only present few of them in Figure 2.

At the variability realization level, JHipster relies on Yeoman template files (JavaScript, Java, HTML, XML, ...) for holding common parts but also properties specific to some variants. Conditional compilation is the main implementation mechanism for realizing variability. With Yeoman templates, some specific code in the different artefacts is activated depending on user's configuration. A first example is given in Listing 3 for Maven files: `hikaricp.version` Maven property is defined only if the configuration includes an SQL database. A second example in given in Listing 4 for Java files. The method *h2TCPServer* is only used in configurations relying on H2 databases (either h2Disk or h2Memory). The inheritance of AbstractMongoConfiguration depends on the activation of *mongodb*. Java annotations are also subject to variations. Thus, variability information is scattered in different artefacts (e.g., Maven, Java, JavaScript, *etc.*).

2.1 Analysis Workflow

From the feature model, we devised an automated way to generate JHipster's variants in order to check their validity (are the variants correctly generated? do the Web-apps compile? Can we build them?). This was done by building for each variant the matching `.yo-rc.json` file and then calling the generator (*yo jhipster*) on each of them. The

[4]The complete list is available on https://jhipster.github.io/
[5]For this study, we use JHipster v3.6.1: https://github.com/jhipster/generator-jhipster/releases/tag/v3.6.1

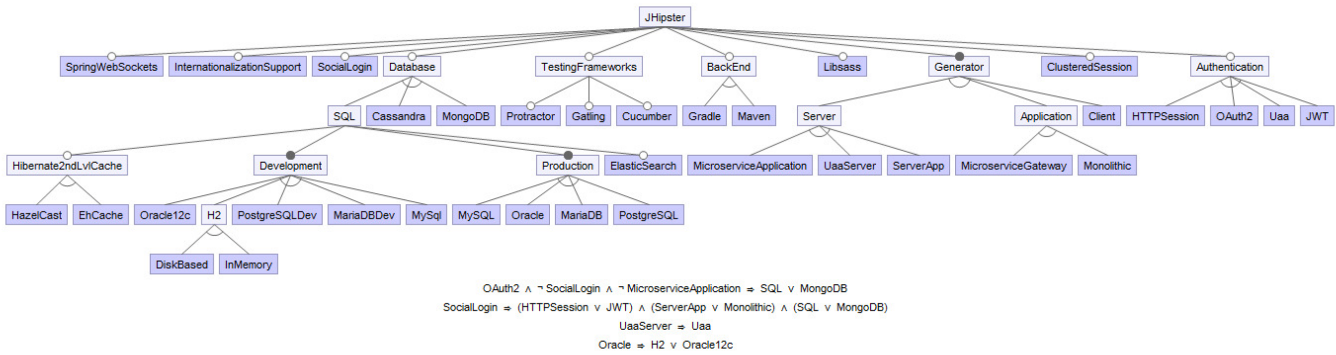

Figure 2: JHipster reverse engineered feature model

Listing 3: _pom.xml excerpt

```
<?xml version="1.0" encoding="UTF-8"?>
(...)
  <properties>
        <%_ if (databaseType === 'sql') { _%>
        <hikaricp.version>2.4.6</hikaricp.version>
        <%_ } _%>
        <awaitility.version>1.7.0</awaitility.version>
        (...)
  <properties>
(...)
```

Listing 4: _DatabaseConfiguration.java Excerpt

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

keys Yeoman expects to find in the `.yo-rc.json` file can be found in the function *saveConfig* of JHipster's `index.js` files. From there we can then run variant dependent commands (Maven or Gradle, Docker, ...) to compile, build and test them (see Figure 4). For each variant, we store some interesting information in a CSV file to analyse later on. Currently, this information is the result of the generation/-compilation/build processes; the logs from each process if there is a problem; the duration of the generation, compilation, and build phases; the size of the Docker image; and the results from the unit tests. This analysis workflow currently runs on a subset of JHipster's variants: we do not, yet, generate client or server standalone application variants (which may be obtained using JHipster sub-generator); we exclude variants with an external databases (Oracle or H2); and we include all test frameworks in each variant (if the constraints allow it), as it prevents the generation of similar variants with only different test frameworks. These choices were motivated both by technical reasons (Oracle being a proprietary database additional work is needed to use it properly) and practical reasons (What can we test with client only app? Are client/server parts not tested in the other types of application?). This selection decreased the total number of variants to about 4600.

2.2 Preliminary Analyses

With the analysis workflow presented in Figure 4, we have already tested the validity (generation, compilation, build) of about 300 variants and found unreported bugs (*i.e.*, anything that would prevent the generation, compilation, or execution of a variant of JHipster, or lead to deviant behaviour) in a few of them. For example, the first bug we found is related to the Docker image of the MariaDB database, in monolithic applications, and it is encountered while trying to deploy the application via Docker. Basically, Docker is looking for the wrong repository/tag, one that doesn't exist. The cause of this error is a missing line in the file `/src/main/docker/app.yml`: a condition *prodDatabaseType == 'mariadb'* on line 5. This issue is still found in JHipster 3.9.1 and doesn't seem to have been detected (currently no related issue post mention it). We will extend the number of tested variants to possibly all of them. We will also use the 3 supported testing frameworks (Cucumber, Protractor and Gatling) to evaluate beyond the correctness of the applications their non-functional properties (performance testing, UI testing, *etc.* see Section 3).

We started investigating preliminary results. For instance, the correlation between generation time and the type of the application (see Figure 3). We observe on this boxplot that micro-service applications and UAA servers require shorter generation times than monolithic applications or micro-service gateways. Indeed, micro-service applications and UAA servers do not need the client part of JHipster's applications. We hope to extract information regarding non-functional properties of the generated Web-apps.

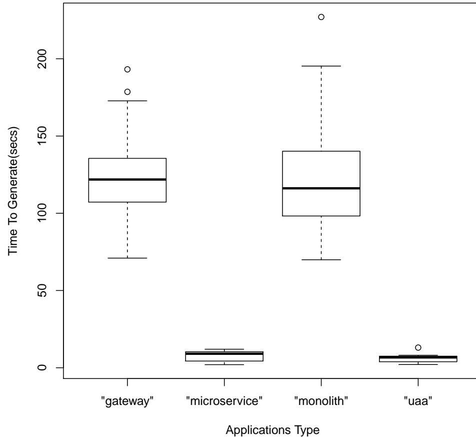

Figure 3: Distribution of generation time by application type boxplot

3. A CASE FOR RESEARCHERS

In the previous section, we presented our derivation infrastructure as well as a few statistics on the generated products and issues found. In this section, we explore two vertices of the "PLA cube" [51]: product-based and family-based analyses. We explain why JHipster is a good candidate to devise new techniques and perform additional empirical assessment of existing ones.

3.1 Products' Analyses

Product lines usually allow a large number of products. Two approaches are possible to validate them. The use of formal methods which prove correctness properties in the specification at the product line level such that all derived products satisfy the same properties, without needing to enumerate all of them [49, 5]. Another approach is to rely on testing, which main goal is to select and sort the fittest set of products to test according to given criteria in order to detect as much bugs as possible. Systematic studies show that a lot of effort has been put on SPL testing [8, 13, 32].

3.1.1 Structural Sampling

Sampling techniques. To reduce the number of products to test, one popular research direction is to use Combinatorial Interaction Testing (CIT) techniques [6, 31] and pairwise (generalized to t-wise) criteria [28, 30, 33, 40]. Over the years, several tools have been developed and support pairwise based selection on the feature model, e.g., [18, 22]. In order to support larger t values, as well as larger feature models, other search-based heuristics have been proposed [3, 17, 45, 37]. All of those CIT, t-wise, and other search-based techniques make the hypothesis that bugs come from interactions between few features and try to select an adequate set of products to test in order to cover as much feature combinations as possible. They have been extensively validated on a large number of feature models, with different sizes, and coming from different sources. However, very few evaluations have actually built the set of products to test in their process.

JHipster potential for Sampling. With $\approx$ 163,000 possible products, JHipster is both non-trivial (as opposed to some academic models in the SPLOT repository) without being as large a Linux, WordPress or Drupal cases. This particularity makes accessible the generation of ***all*** the variants. The idea is to be able to obtain a *ground truth* to compare the efficiency of sampling algorithms. By being able to compute absolute values for the numbers and types of interaction bugs, biases when assessing techniques can be reduced. As noted by Jin *et al.* [21], configuration issues can happen everywhere: we believe that the variety of technologies at work in a JHipster derived product is also an opportunity to study such aspects. As seen in Section 2, our feature model integrates variability information from different files and will lead researchers to study different kinds of interaction bugs.

3.1.2 Functional Testing

Product-level functional testing. As noted by Von Rhein *et al.* [51], product-based analysis strategy is simple: we analyse each product individually without taking into account variability (it has been resolved using sampling or enumerating all products). The benefit is that single-product analysis tools can be used. Researchers have proposed to derive test cases from product line scenarios and use cases (e.g., [34]) promoting the reuse of test models and artefacts.

JHipster's potential. As opposed to Drupal or Wordpress cases, where test cases are either optional or solely depending on the will of plugin developers [43, 36, 15], JHipster comes with a systematic testing infrastructure and test cases are deployed for every Web-app deployed. In particular, Cucumber [7] supports early testing in the form of scenarios. Integration with code coverage tools is also available. However our preliminary analyses shown that the provided tests were quite simple, product-agnostic (based on a generic application that is the root of all products) and code coverage was quite low. Thus, we should derive test cases that take into account the specificities of each product, to get a better *base coverage* prior to the development of a richer Web-app on top of the derived product.

3.1.3 Non-Functional Analyses

Feature-related quality attributes. Recent research shifts from functional validation using testing to detect undesired feature interactions to non-functional analyses in order to predict performance of a given product [44, 47, 48, 46]. Using statistical learning [16] and regression methods [50], or mathematical models to predict and detect (undesired) performance-relevant feature interactions [53].

JHipster potential for quality analyses. Web-apps are particularly interesting cases for performance, since this quality attribute has a direct influence on Websites' successes. To this end, JHipster comes with Gatling[6], a load testing tool. It is possible to experiment with feature-related performance techniques in order to assess the proposed theories and calibrate statistical learning. Note that performance is not the only quality attribute that can be studied: security is also key especially for e-commerce websites. While the JHipster infrastructure does not currently offer any security-dedicated analysis toolset, the diversity of technologies used in a JHipster application and our automated derivation approach allow to focus on a given technology in various security scenarios.

[6]http://gatling.io/

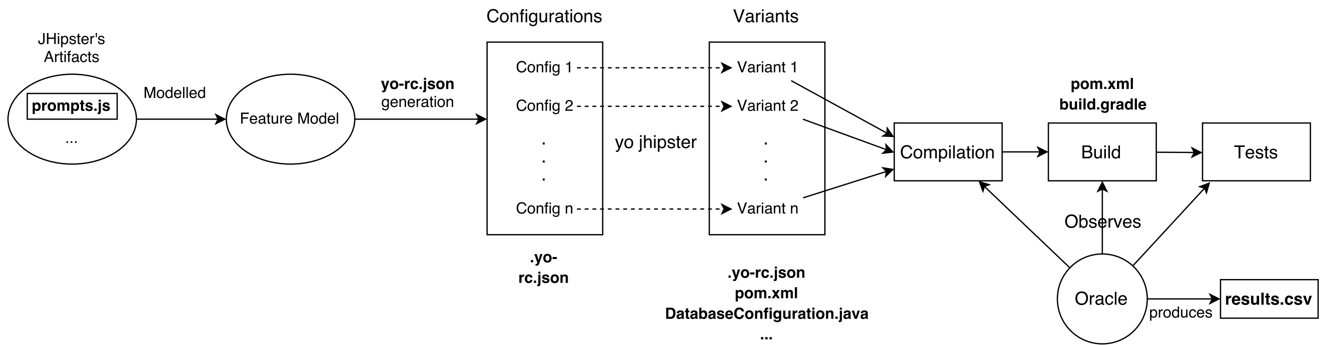

Figure 4: Complete Analysis Workflow

3.2 Family-based Analyses

Variability-aware techniques. So far, our infrastructure focuses on the product level by sampling products of interest and analysing them individually using provided validation environments (Protactor, Karma.js, Cucumber, etc.). While our goal is to obtain a ground truth by analysing all the variants [14], exploiting variability to reuse analyses (e.g. tests, proofs, etc.) in order to reduce the overall analysis effort and better scale large cases is relevant [27]. Model-based testing approaches use behavioural models of the product line to generate test cases for the different products: (resp.) delta-oriented product line testing [29, 26] and featured transition system based [10, 9, 11] approaches use (resp.) state machines and transition systems in order to capture the common and product specific behaviour of the product line. At the code level, variability-aware parsers [24], variational structures [52] and type-checking [23] are of interest. They enable *variability-aware testing* [25] to, for example, evaluate a test case against myriads of configurations in one run [36].

JHipster potential. JHipster offers an interesting playground for the aforementioned analyses. The difficulty for model-based approaches is to obtain accurate models of the case study when the implementation already exists. To this end, server execution logs of different variants can be used to extract part of the system behaviour [9]. At the code level, the challenge is to specify annotations across multiple technological spaces, while performing commonality and variability analysis. JHipster provides a generic customisable Web-app for each variant, enabling a user to log in and to create entities, as a starting point. JHipster also includes a configurator, allowing to consider the interaction between configuration workflows [19] and derived products [39].

3.3 Product Line Evolution

Evolution techniques. From the evolution perspective, product lines represent an interesting challenge. Product lines developers have to manage updates at different levels: the evolution of the variability model and the mapping to other artefacts [12]; the evolution of the artefacts themselves which will impact several products [35, 42]; and the evolution of the configurator and configuration workflow. To understand how existing SPL are updated, Passos *et al.* [38] recently studied the Linux kernel variability models and other artefact types co-evolution.

JHipster potential. With 146 releases since 2013, JHipster is under active development and evolution. Therefore a challenge for researchers is to devise automated means to update the JHipster feature model. As opposed to the Linux case, where part of the variability model can be extracted from KConfig, several JavaScript files are necessary to build it, pushing for more versatile variability inference techniques.

4. A CASE FOR EDUCATION

JHipster has been used as part of different teaching courses. In this section, we report on such an experience and then argue that JHipster is a relevant case for education and in particular for SPL teaching.

4.1 Experiences

Experience #1. Our first teaching experience with JHipster started in 2015 at University of Rennes 1. The audience was 40+ MSc students with a speciality in software engineering or in software management. As part of a model-driven engineering course, we used to teach variability modelling and implementation techniques. In 2015, we decided to slightly change the way variability is explained and we notably introduced JHipster, for the following reasons.

First, students used JHipster in another course dedicated to Web development. Therefore students could reuse JHipster for building a quite complex Web application in the model-driven engineering course: a Web generator of video variants called *VideoGen*. *VideoGen* is a software application that builds video variants by assembling different video sequences; it is a generalization of a real-world Web generator [4]. Video variants can be randomly chosen or users can configure their videos through a Web interface. A textual specification, written in a domain-specific language, documents what video sequences are mandatory, optional, or alternatives. Frequencies and constraints can also be specified[7]. *VideoGen* challenges students to master Web development as well as variability modelling and implementation techniques: they should build a Web configurator, implement algorithms for randomly choosing and building a video variant, etc. The video generator was the running example of the course and was used in the lab sessions and in the project for evaluating students. JHipster was used all along

[7]More details can be found online: https://github.com/FAMILIAR-project/teaching/tree/gh-pages/resources/Rennes2015MDECourse

to implement the Web application, including a Web configurator. In summary, JHipster was used as a relevant technology for showing the relations with other courses (Web development) and for implementing a non-trivial variability system (a video generator) based on modelling technologies.

The second reason is that we took the opportunity to explain *how* JHipster is implemented and more precisely how variability concepts and techniques are applied in practice. During the course, we used JHipster to define what a software product line is, making the correspondences with other well-known configurable systems like Linux, Firefox, or ffmpeg. We explained variability implementation techniques and in particular conditional compilation, templates and annotative-based approaches with the use of JHipster. From a variability modelling perspective, we introduced feature models by using the configurator of JHipster. In the lab sessions, students used the JHipster generator to obtain a Web stack and develop the video generator. They had to make the Web video generator configurable, for instance, they had to implement the ability to save or not a video variant. We have also proposed different exercises related to feature modelling. Along the way, students could exercise on variability concepts that were also found in JHipster.

Our experience was mostly positive. The evaluation of the students' projects on the Web video generator gives high marks. Interactions with students during the courses show that JHipster helps to understand more concretely variability concepts. However we noticed two limitations. First, students manipulated variability concepts at two levels and for two different purposes. The first level was for creating from scratch a configurable video generator, involving skills in domain-specific languages, model transformations, and variability modeling. The second level was for understanding and reusing the JHipster generator. There was some confusions between the two levels. The explanations on JHipster certainly deserve more time and a specific attention – perhaps a dedicated exercise, see hereafter. Second, the technology behind JHipster is quite advanced and requires numerous skills. Some students have technical difficulties to connect the dots and transfer their conceptual knowledge into concrete terms. We had to postpone the deadline for project delivery to let students enough time to master the Web stacks.

For mitigating the two weaknesses, we have decided in 2016 to play the full course on Web development *before* the model-driven engineering and variability courses. We expect that students can, prior to the course, master JHipster for (1) better understanding its internals; (2) better implementing the variability concepts.

Experience #2. Our second teaching experience with JHipster was in late 2015 at University of Rennes 1. This time the audience was MSc students with a strong interest in research. We reused almost the same material as previously but we also addressed more advanced topics like automated reasoning with solvers, software product line verification and validation, *etc.*. We used JHipster for the same previous reasons. Compared to the first experience, JHipster was the sole focus of this course and there was no video generator to develop. It simplified how variability concepts were introduced and explained. The project aimed at evaluating students and was oriented for addressing some open research questions: How to elaborate and reverse engineer a feature model of the JHipster generator? What are the configuration bugs of JHipster? How to automatically find those bugs?

With the JHipster case, students could elaborate a feature model based on a static analysis of several artefacts. They could apprehend the combinatorial explosion inherent to variability-intensive systems. Overall students could revisit the variability techniques of the course with a realistic and complex example. The JHipster case also shows to students the connection with other research works, mainly what we have described in the previous section. Some questions were voluntary open like the proposal of "a strategy for testing the configurations of JHipster at each commit or release".

Another motivation for us was to use JHipster to explore some research directions and make some progress with students. We asked them to collect and classify configuration bugs on GitHub. We also gathered several feature models based on their analysis. Such works help us to re-engineer JHipster as a software product line: we reused such feature models to initiate the work exposed in Section 2. Students of the course did not design or develop the workflow analysis of Figure 4. Discussions and insights, however, motivate the need to build such a testing infrastructure for JHipster.

Overall, the work of students was evaluated in a positive way. They demonstrated their abilities to understand and use variability concepts. It was also useful for our own research work.

Other experiences. We have used JHipster in other educating settings in 2015: *(a)* at University of Montpellier for MSc students; *(b)* within the DiverSE Inria team for forming PhD students to variability. Such experiences deserve less comments since the duration of the courses was one full-day. Yet the JHipster case was again useful to us, educators and researchers, to both illustrate the variability concepts and exchange on open issues.

4.2 JHipster for Education

A survey on teaching of software product lines showed that two recurring issues for educators are the absence of case studies and the difficulty to integrate product lines within a curriculum [2]. Similar concerns have been raised at SPLTea'14 and SPLTea'15 workshops (see http://spltea.irisa.fr). JHipster acts as an interesting and useful case for addressing these issues. Specifically, JHipster can be used for: (i) illustrating a product line course and for describing variability modelling and implementation techniques with a real-world case over different technologies; (ii) conducting lab sessions in relation with variability; (iii) connecting or better integrating product line courses to other courses (*e.g.*, Web development, model-driven engineering); (iv) exploring open research directions with students.

In conclusion, our experiences with JHipster were mostly positive, though some improvements can be made. All material (slides, instructions of lab sessions) can be found online: http://teaching.variability.io/. We are reusing the same case and material in 2016 at the University of Rennes 1.

5. CONCLUSION AND FUTURE WORK

In this paper, we described JHipster as a case for experimenting with various kinds of variability-related analyses and teaching software product lines. We introduced an analysis workflow that automates the derivation of JHipster variants (Web-apps) on the basis of a feature model manually

extracted from Jhipster questionnaire's files. As the number of possibilities is within reach of current (distributed) computing facilities, some "all-products" information may be obtained, which is useful to assess some specific techniques such as sampling. Our analysis workflow is also relevant for education to understand and explore product line derivation testing and analysis concepts. Our analysis infrastructure is only in its premises and naturally calls for future developments. At the research level, we would like of course to share the results obtained on running analyses on the whole product line. This requires running our workflow on distributed infrastructure like Grid5000 (https://www.grid5000.fr/), an option that we are currently studying. We also want to share our results (bugs, performance issues) with the JHipster developers so that they can take advantage of them in their fixes and releases. We finally would like to introduce this workflow in our SPL teaching curriculum and continue to share it openly with the community.

6. ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their feedback. This work was partly supported by the European Commission (FEDER IDEES/CO-INNOVATION).

7. REFERENCES

[1] M. Acher, P. Collet, P. Lahire, and R. B. France. FAMILIAR: A domain-specific language for large scale management of feature models. *Science of Computer Programming (SCP)*, 78(6):657–681, 2013.

[2] M. Acher, R. E. Lopez-Herrejon, and R. Rabiser. A survey on teaching of software product lines. In *VaMoS '14*, Nice, France, jan 2014. ACM.

[3] M. Al-Hajjaji, S. Krieter, T. Thüm, M. Lochau, and G. Saake. IncLing: efficient product-line testing using incremental pairwise sampling. In *GPCE '16*, pages 144–155. ACM, 2016.

[4] G. Bécan, M. Acher, J.-M. Jézéquel, and T. Menguy. On the variability secrets of an online video generator. In *Variability Modelling of Software-intensive Systems (VaMoS'15)*, pages 96 – 102, Hildesheim, Germany, jan 2015.

[5] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. J.-F. Raskin. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Transactions on Software Engineering*, 39(8):1069–1089, aug 2013.

[6] M. Cohen, M. Dwyer, and Jiangfan Shi. Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. *IEEE Transactions on Software Engineering*, 34(5):633–650, 2008.

[7] CucumberTeam. Cucumber website. https://cucumber.io, accessed in November 2016.

[8] P. A. da Mota Silveira Neto, I. do Carmo Machado, J. D. McGregor, E. S. de Almeida, and S. R. de Lemos Meira. A systematic mapping study of software product lines testing. *Information and Software Technology*, 53(5):407–423, 2011.

[9] X. Devroey, G. Perrouin, M. Cordy, H. Samih, A. Legay, P.-Y. Schobbens, and P. Heymans. Statistical prioritization for software product line testing: an experience report. *SoSyM*, pages 1–19, 2015.

[10] X. Devroey, G. Perrouin, A. Legay, M. Cordy, P.-Y. Schobbens, and P. Heymans. Coverage Criteria for Behavioural Testing of Software Product Lines. In *ISoLA '14*, volume 8802 of *LNCS*, pages 336–350. Springer, 2014.

[11] X. Devroey, G. Perrouin, A. Legay, P.-Y. Schobbens, and P. Heymans. Search-based Similarity-driven Behavioural SPL Testing. In *VaMoS '16*, pages 89–96. ACM, 2016.

[12] N. Dintzner, A. van Deursen, and M. Pinzger. FEVER: Extracting Feature-oriented Changes from Commits. In *MSR '16*, pages 85–96. ACM, 2016.

[13] E. Engström and P. Runeson. Software product line testing - A systematic mapping study. *Information and Software Technology*, 53(1):2–13, 2011.

[14] J. A. Galindo, M. Acher, J. M. Tirado, C. Vidal, B. Baudry, and D. Benavides. Exploiting the Enumeration of All Feature Model Configurations. In *SPLC '16*, Beijing, China, Sept. 2016.

[15] M. Greiler, A. van Deursen, and M. A. Storey. Test confessions: A study of testing practices for plug-in systems. In *ICSE '12*, pages 244–254. ACM, 2012.

[16] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski. Variability-aware performance prediction: A statistical learning approach. In *ASE '13*, pages 301–311. IEEE, 2013.

[17] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. Le Traon. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines. *IEEE Transactions on Software Engineering*, 40(7):650–670, 2014.

[18] A. Hervieu, B. Baudry, and A. Gotlieb. PACOGEN: Automatic Generation of Pairwise Test Configurations from Feature Models. In *ISSRE '11*, number i, pages 120–129. IEEE, 2011.

[19] A. Hubaux, A. Classen, and P. Heymans. Formal modelling of feature configuration workflows. In *SPLC '09*, pages 221–230. Carnegie Mellon University, 2009.

[20] JHipsterTeam. Jhipster website. https://jhipster.github.io, accessed in November 2016.

[21] D. Jin, X. Qu, M. B. Cohen, and B. Robinson. Configurations everywhere: implications for testing and debugging in practice. In *ICSE '14 Companion Proceedings*, pages 215–224. ACM, 2014.

[22] M. F. Johansen, Ø. Haugen, and F. Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *SPLC '12*, volume 1, page 46. ACM, 2012.

[23] C. Kastner and S. Apel. Type-checking software product lines-a formal approach. In *ASE '08*, pages 258–267. IEEE, 2008.

[24] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *ACM SIGPLAN Notices*, volume 46, pages 805–824. ACM, 2011.

[25] C. Kästner, A. Von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann. Toward

variability-aware testing. In *FOSD '12*, pages 1–8. ACM, 2012.

[26] R. Lachmann, S. Lity, S. Lischke, S. Beddig, S. Schulze, and I. Schaefer. Delta-oriented test case prioritization for integration testing of software product lines. In *SPLC '15*, pages 81–90. ACM, 2015.

[27] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *ESEC/FSE '13*, pages 81–91. ACM, 2013.

[28] M. Lochau, S. Oster, U. Goltz, and A. Schürr. Model-based pairwise testing for feature interaction coverage in software product line engineering. *Software Quality Journal*, 20(3-4):567–604, 2012.

[29] M. Lochau, I. Schaefer, J. Kamischke, and S. Lity. Incremental Model-Based Testing of Delta-Oriented Software Product Lines. In *TAP '12*, volume 7305 of *LNCS*, pages 67–82. Springer, 2012.

[30] R. E. Lopez-Herrejon, F. Chicano, J. Ferrer, A. Egyed, and E. Alba. Multi-objective Optimal Test Suite Computation for Software Product Line Pairwise Testing. In *ICSME '13*, pages 404–407. IEEE, 2013.

[31] R. E. Lopez-Herrejon, S. Fischer, R. Ramler, and A. Egyed. A first systematic mapping study on combinatorial interaction testing for software product lines. In *ICSTW '15*, pages 1–10. IEEE, 2015.

[32] I. d. C. Machado, J. D. McGregor, Y. C. Cavalcanti, and E. S. de Almeida. On strategies for testing software product lines: A systematic literature review. *Information and Software Technology*, 56(10):1183–1199, 2014.

[33] D. Marijan, A. Gotlieb, S. Sen, and A. Hervieu. Practical pairwise testing for software product lines. In *SPLC '13*, page 227. ACM, 2013.

[34] C. Nebut, Y. L. Traon, and J. Jézéquel. *System Testing of Product Lines: From Requirements to Test Cases*, pages 447–477. Springer, 2006.

[35] L. Neves, P. Borba, V. Alves, L. Turnes, L. Teixeira, D. Sena, and U. Kulesza. Safe evolution templates for software product lines. *Journal of Systems and Software*, 106:42–58, 2015.

[36] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *ICSE '14*, pages 907–918. ACM, 2014.

[37] J. A. Parejo, A. B. Sánchez, S. Segura, A. Ruiz-Cortés, R. E. Lopez-Herrejon, and A. Egyed. Multi-objective test case prioritization in highly configurable systems: A case study. *Journal of Systems and Software*, 122:287–310, 2016.

[38] L. Passos, L. Teixeira, N. Dintzner, S. Apel, A. Wąsowski, K. Czarnecki, P. Borba, and J. Guo. Coevolution of variability models and related software artifacts. *Empirical Software Engineering*, 21(4):1744–1793, 2016.

[39] G. Perrouin, M. Acher, J.-M. Davril, A. Legay, and P. Heymans. A complexity tale: web configurators. In *VACE '16*, pages 28–31. ACM, 2016.

[40] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. le Traon. Pairwise testing for software product lines: Comparison of two approaches. *Software Quality Journal*, 20(3-4):605–643, 2011.

[41] M. Raible. *The JHipster mini-book*. C4Media, 2015.

[42] G. Sampaio, P. Borba, and L. Teixeira. Partially safe evolution of software product lines. In *SPLC '16*, pages 124–133. ACM, 2016.

[43] A. B. Sanchez, S. Segura, and A. Ruiz-Cortes. A Comparison of Test Case Prioritization Criteria for Software Product Lines. In *ICST '14*, pages 41–50. IEEE, 2014.

[44] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki. Cost-Efficient Sampling for Performance Prediction of Configurable Systems (T). In *ASE '15*, pages 342–352. IEEE, 2015.

[45] A. S. Sayyad, T. Menzies, and H. Ammar. On the value of user preferences in search-based software engineering: A case study in software product lines. In *ICSE '13*, pages 492–501. IEEE, 2013.

[46] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner. Performance-influence models for highly configurable systems. In *ESEC/FSE '15*, pages 284–294. ACM, 2015.

[47] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. Predicting performance via automated feature-interaction detection. In *ICSE '12*, pages 167–177. IEEE, 2012.

[48] N. Siegmund, M. Rosenmüller, C. Kästner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. *Information and Software Technology*, 55(3):491–507, 2013.

[49] M. H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. Modelling and analysing variability in product families: Model checking of modal transition systems with variability constraints. *Journal of Logical and Algebraic Methods in Programming*, nov 2015.

[50] P. Valov, J. Guo, and K. Czarnecki. Empirical comparison of regression methods for variability-aware performance prediction. In *SPLC '15*, pages 186–190. ACM, 2015.

[51] A. Von Rhein, S. Apel, C. Kästner, T. Thüm, and I. Schaefer. The pla model: on the combination of product-line analyses. In *VaMoS '13*, page 14. ACM, 2013.

[52] E. Walkingshaw, C. Kästner, M. Erwig, S. Apel, and E. Bodden. Variational data structures: Exploring tradeoffs in computing with variability. In *SPLASH '14*, pages 213–226. ACM, 2014.

[53] Y. Zhang, J. Guo, E. Blais, K. Czarnecki, and H. Yu. A mathematical model of performance-relevant feature interactions. In *SPLC '16*, pages 25–34. ACM, 2016.