UNIVERSITÉ DE NAMUR

Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

JHipster

Analyse de l'impact de l'²volution sur le processus de test

Nguepi Kenfack, Koko

Award date:
2018

Awarding institution:
Universite de Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 23. Apr. 2024

Université de Namur
Faculté d'informatique
Année académique 2017–2018

JHipster : Analyse de l'impact de l'évolution sur le processus de test

NGUEPI Kenfack Koko

Maître de stage : Mathieu ACHER

Promoteur : ________________ (Signature pour approbation du dépôt - REE art. 40)
Heymans Patrick

Co-promoteur : Perrouin Gilles

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Resumé

Le test exhaustif des systèmes logiciels configurables est en pratique assez difficile d'où l'utilisation des techniques d'échantillonnages et d'abstraction de la variabilité afin de réduire l'espace des configurations à tester. Halin *et al.* [1] décrivent une infrastructure permettant le test exhaustif d'un cas particulier de système configurable : le générateur d'application JHipster 3.6.1. Ils décrivent ensuite l'effort d'ingénierie qu'ils ont déployé pour effectuer le test. Dans la suite de leurs travaux, ils définissent 6 classes de fautes permettant de catégoriser les Bugs identifiés par le test exhaustif. Enfin, ils comparent quatre stratégies de test par échantillonnage permettant de réduire l'espace configurations en validant pour chacune d'elle en fonction de la taille de l'échantillon : le nombre de Bugs et le nombre de classe de fautes Identifiées afin de conclure sur les stratégies les mieux adaptées pour détecter les fautes en fonction de la taille de l'échantillon. Face à l'évolution rapide de JHipster, nous aimerons savoir si les travaux de Halin *et al.* permettent de soutenir cette évolution autrement dit, leurs résultats sont-ils encore valides pour les versions suivantes de JHipster ? Telle est la question a laquelle nous essayerons de répondre dans ces travaux. À cet effet, nous commencerons par dupliquer les travaux de Halin *et al.* sur une version plus récente de JHipster(version 4.8.2), possédant plus de fonctionnalitées et donc plus de produits possibles. Ensuite, nous analyserons et comparerons les résultats obtenus, en terme d'effort d'ingénierie pour la reproduction de [1] mais également les observations relatives aux fautes détectées et aux performances des méthodes d'échantillonnage utilisées par Halin *et al.* mais cette fois-ci appliquées sur JHipster 4.8.2. Ceci nous permettra alors d'apporter des élements quant à la généralisation possible des résultats obtenus par Halin *et al.* .

textit **Mots clés :** Varabilité logicielle ; JHipster ; Lignes de produits logiciels ; Test des Lignes de Produits Logiciels ; Test combinatoire d'interaction ; Évolution des Lignes de Produits Logiciels ; T-wise ; PLEDGE, IGRIDA ; Échantillonnage

Abstract

Exhaustive testing of configurable software systems is in practice quite difficult, Therefore, we use sampling technique and abstraction of variability in order to reduce the space of the configurations to be tested become a good practice. Halin *et al.* describe an infrastructure allowing to exhaustively test a particular case of configurable system : the JHipster 3.6.1 application generator. They describe the engineering effort they have deployed to test JHipster. They also compare four sampling test strategies to reduce the configuration space by validating for each of them the number of identified faults compared to the exhaustive test, in order to conclude on the most appropriate strategies for detecting faults depending of the size of the sample. Given the constant evolution of JHipter, we would like to know if the work of Halin *et al.* can support this evolution : in other words, are their results still verifiable with another recent versions of JHipster ? This is the question we are trying to answer in this work. For this purpose, we duplicate Halin *et al.* work on a more recent version of JHipster (version 4.8.2), having more functionalities and therefore more products. Then we analyze and compare the results obtained, in terms of engineering effort for the reproduction of [1] but also the observations relating to the detected faults and the performance sampling methods used by Halin *et al.* but this time applied on JHipster 4.8.2. This allowed us to provide insights on the generalization of the results obtained by Halin *et al.*

Keywords : Software Variability ; JHipster ; Software product lines ; Testing Software Product Lines ; Combinatorial interaction test ; Software Product Lines evolution ; IGRIDA ; T-wise ; PLEDGE

Préface

Ce mémoire est rédigé par Koko Nguepi Kenfack dans le cadre de sa dernière année d'étude en informatique à l'Université de Namur.

Cette thèse est le produit d'une période d'étude de deux années de Master incluant un stage à l'INRIA Rennes-Bretagne Atlantique, dans l'équipe DiverSE, sous la supervision de mes promoteurs Patrick Heymans et Gilles Perrouin et mon maitre de stage Mathieu Acher (Université de Rennes 1) du 1 octobre 2017 au 30 janvier 2018.

Tout d'abord, je voudrais souligner et remercier la remarquable supervision de tous mes superviseurs (Gilles et Patrick) et de mon maitre de stage (Mathieu). Je remercie également l'équipe DiverSE pour son accueil chaleureux et les nombreuses discussions, conseils et soutiens qu'il m'a apporté durant mon stage. Sans oublier Xavier Devroey qui m'a également apporté une aide indéniable à un moment critique de ces travaux, Halin Axel et Nuttinck Alexandre qui ont toujours répondu à mes préoccupations et m'ont guidé sur certain aspect de ce travail.

Je voudrais terminer en remerciant mes amis et ma famille de m'avoir gardé motivé pendant tout ce temps. Merci Willy Koko et Mdme pour votre soutien. Merci Benita toi qui a toujours été là pour moi.

Glossaire

Build Utilisé pour désigner la phase de construction d'une application, elle consiste en : la récupération des bibliothèques et des ressources nécessaires à l'exécution de l'application mais aussi à la construction de l'exécutable. 41

Erreur Action ou une omission pouvant introduire des fautes dans le système. 38

Failure C'est un effet indésirable observé dans le comportement externe d'un système. C'est donc la manifestation de l'echec d'un système. 30

faute C'est une source d'échec du système. Elles produisent des Failures qui sont obversable dans le comportement externe du système. 38

Feature C'est une fonctionnalité offerte par un système ou une caractéristique que possède ce dernier. 41

Feature Model Encore appelé modèle caractéristique, c'est un modèle permettant de modéliser la variabilité d'un système configurable. 2, 17

LdPL Famille de systèmes qui partagent un ensemble commun d'actifs techniques de base, avec des extensions et des variantes pré-programmées pour répondre aux besoins de clients ou de segments de marché spécifiques [5]. 2, 3, 10, 16–22

Workflow de test Désigne la procedure étape par étape élaborée pour tester les différentes configurations de JHipster et récolter les resultats . 4, 45

Table des matières

Resumé **i**

Abstract **iii**

Préface **iv**

1 Introduction **10**

2 État de l'art **13**

2.1 Les lignes de produit logiciel (LdPL) 13

2.1.1 Définition et Exemple 14

2.1.2 Ingenierie des lignes de produits logiciels 14

2.2 Modélisation de la variabilité et derivation des produits d'une LdPL . . . 16

2.2.1 Notion de variabilité 16

2.2.2 Modélisation de la variabilité d'une LdPL 16

2.2.2.1 Le Feature Model 17

2.2.3 La dérivation des produits 18

2.2.3.1 Dérivation par configuration 19
2.3 Évolution des LdPLs 19
2.3.1 Une approche d'évolution des lignes de produits logiciels 21
2.3.1.1 Approche d'évolution par différences de Modèles 22
2.3.1.2 Approche d'évolution par opérateur de changement 23
2.4 Test des produits d'une lignes de produits logiciels 23
2.4.1 Approche en force 24
2.4.2 Approche par sélection des cas de tests 24
2.4.2.1 Sélection par interaction combinatoire 25
2.4.2.2 Échantillonage basé sur la dissimilarité 27
2.4.2.3 Échantillonage aléatoire(Random sampling) 27
2.4.2.4 Échantillonage par activation/désactivation de Feature . . 28
2.4.3 Approche par utilisation des cas de test 30
2.4.3.1 Minimisation des suites de test 30
2.4.3.2 Priorisation des cas de test 30

3 Cas d'étude : JHipster 32

3.1 Test de JHipster 32
3.1.1 JHipster 32
3.1.2 Tester JHipster 34
3.1.2.1 Extraction de la variabilité et dérivation des configurations 35

3.1.2.2 Génération,Compilation et Exécution des configurations . 35
3.1.2.3 Exécution du Workflow 37
3.1.2.4 Les résultats obtenus 37
3.2 Évolution de JHipster 38
3.3 Motivation 39
3.4 Méthode et Question de recherche 40
3.4.1 Questions de recherche 40
3.4.2 Méthodologie 41
3.4.2.1 Extraction de la variabilité 41
3.4.2.2 Dérivation des configurations associées au Feature Model . 43
3.4.2.3 Mise à jour de l'infrastructure de test 43
3.4.2.4 Analyse du Workflow de test et correction des Bugs 45
3.4.2.5 Déploiement et exécution du Workflow sur un environnement de Calcul Distribué 49

4 Resultats **54**
4.1 Le coût du test 55
4.1.1 Effort d'ingénierie 55
4.1.2 Coût en ressource de calcul 56
4.2 Analyse des Bugs 58
4.2.1 Analyse statistique quantitative 58
4.2.2 Analyse statistique qualitative 60

4.2.3 Analyse de résultat de sampling 64
4.3 Analyse comparative 66
4.3.1 L'éffort d'ingénierie 66
4.3.2 Taux et types de bugs 68
4.3.3 Efficacité des méthodes de sampling 69

5 Conclusion **71**

Bibliography **73**

Appendices **78**

A JHipster 4.8.2 : description des Technologies utilisées **79**
A.1 Types d'applications générés 79
A.2 Les modes d'authentifications 80
A.3 Les types de base de données 81
A.4 Les Frameworks de test 82
A.5 Autres Frameworks 82

B Description du data set JHipster 4.8.2 **84**
B.1 Description des entrées du CSV 85

C Script et algorithmes utilisés **87**
C.1 Analyse quantitative et qualitative 88

C.1.1 Graphique présentant les proportions d'erreurs par Feature 88

C.1.2 Graphique présentant les proportions d'erreurs par classe de faute . 89

C.1.3 Génération des Règles d'associations pour les erreurs de compilation 90

C.1.4 Génération des Règles d'associations pour les erreurs de Build . . . 90

C.2 L'échantillonnage 91

C.2.1 Déterminer l'efficacité d'une stratégie de d'échantillonnage pour la détection des fautes 91

C.2.2 Déterminer l'efficacité d'une stratégie de d'échantillonnage pour la détection d'erreurs 92

Liste des tableaux

3.1 Feature et évolution de JHipster 39

4.1 Règles d'association généralisées pour les échecs de compilation et de Build 61

4.2 Résultat des strategies d'échantillonnage 65

4.3 Comparaison : coût du test 67

4.4 Comparaison : quantité et qualité des Bugs 68

A.1 Description des types d'applications 80

A.2 Description des modes d'authentification 81

A.3 Description des Framework de test 82

A.4 Description des autres frameworks 83

B.1 Description du fichier jhipster.csv 85

B.2 Description des modes d'authentification 86

Table des figures

2.1 Processus d'ingenierie des lignes de produits logiciels 15

2.2 Exemple de Feature Model de decrivant l'e-shop 18

2.3 Exemple de configuration de l'e-shop 19

2.4 Evolution d'une LdPL par Feature Model 22

2.5 Différence de Modèle pour l'évolution 23

2.6 Selection des cas de tests 24

2.7 Description de pairwise 26

2.8 Feature Model illustratif de la technique One-Disabled 29

3.1 Interface de configuration de JHipster 33

3.2 Processus de test des configurations de JHipster 34

3.3 Exemple de configuration décrit par le fichier .yo-rc.json 36

3.4 Workflow de test de JHipster 37

3.5 Exemples de contraintes 42

3.6 Première version du Feature Model. 44

3.7 Procédure d'exécution manuelle des activités de tests 46

3.8 Feature Model corrigé. 48

3.9 Nouveau Workflow de test. 49

3.10 Caractéristiques de quelques machines d'IGRIDA [35] 50

3.11 Déploiement du Workflow sur IGRIDA 53

3.12 Exécution du workflow sur IGRIDA 53

4.1 Proportions d'échec de Build par feature 59

4.2 Proportions d'échec par classe de fautes 62

C.1 Script : Extraction du taux d'erreurs pour chaque Feature 88

C.2 Script : Extraction du taux d'erreurs pour chaque classe de faute 89

C.3 Déterminer les règles d'associations pour les erreurs de compilation 90

C.4 Déterminer les règles d'associations pour les erreurs de Build 90

C.5 Script : Détection de fautes 91

C.6 Script : Détection d'erreurs 92

C.7 Exemple d'appel des deux fonctions 93

Chapitre 1

Introduction

L'idée des lignes de produits logiciels (LdPL) est née par adaptation de celle de la réutilisation de masse instaurée dans les années 1900 par Henri Ford dans le secteur automobile. Ce paradigme vise à réaliser une production de masse de produits logiciels tout en offrant la possibilité d'adapter chaque produit aux besoins courants d'une catégorie d'utilisateurs ciblés. Pour pouvoir répondre à ce besoin de personnalisation, les LdPL sont des systèmes hautement configurables, ils offrent de nombreuses fonctionnalités pouvant être activées ou désactivées en fonction des besoins de l'utilisateur ciblé.

Cependant, soutenue par des activités de développement, de test et de maintenance, l'ingénierie des lignes de produits logiciels fait face à un challenge lié au coût important de leurs réalisations. En effet les LdPL sont des systèmes hautement configurables et donc à forte variabilité, ceci implique un nombre élevé de produits dérivés obtenus par combinaison des différentes fonctionnalités ou options. Un défi majeur pour les développeurs d'une LDPL est alors de s'assurer que toutes les combinaisons d'option (configurations) créent le produit décrit, compilent et exécutent correctement. Le noyau Linux, par exemple, offre plus de 14 000 options (Gazzillo, 2015), le générateur d'application web JHipster version 3.6.1 quant à lui offre 26000 configurations [1] : comment garantir que toutes ces configurations sont correctes (i.e., créent le produit, compilent et exécutent correctement) ?

Ce défi est d'autant plus critique que les défaillances d'un produit peuvent avoir un impact sur l'appréciation des utilisateurs pour une gamme et entraver le succès ou la réputation d'un projet. On arrive alors à se poser la question de savoir comment tester efficacement un système hautement configurable en tenant compte du nombre de configurations dérivées et en limitant les coûts ? Une première idée serait de tester toutes les configurations possibles du système (test exhaustif) mais cette stratégie est en pratique difficile à réaliser. De plus, elle n'atténue pas le problème de coût de test en ce sens qu'elle requiert un effort d'ingénierie et une quantité de ressources considérables [1]. Une autre pratique courante consiste à exécuter et à tester un échantillon de variantes représentatives, énumérant toutes les configurations perçues comme impossibles (susceptible de produire des fautes).

Pour répondre à la question de savoir laquelle des stratégies de test par échantillonnage est la plus adaptée pour tester de tels systèmes, Halin *et al.* [1] ont choisis de comparer les stratégies d'échantillonages face la vérité terrain obtenue par le test exhaustif d'un exemple de système configurable nommé JHipster. En effet, JHipster est un projet open source lancé en octobre 2013, c'est un générateur de code populaire pour les applications Web. Il offre la génération de piles technologiques complètes composées de code d'amorçage Java et Spring du côté serveur et Angular et Bootstrap du côté frontal. Halin *et al.* [1] ont choisi JHipster pour plusieurs raisons. Tout d'abord, il s'appuie sur une variété de langages et de technologies avancées pour étoffer les applications Web et, par conséquent, introduit la variabilité à différents niveaux. Deuxièmement, JHipster propose pour la version étudiée (3.6.1) 48 options de configurations et 15 contraintes entre les options, ce qui conduit à plus de 150 000 configurations [1] ce qui rend la taille du système gérable. Halin *et al.* [1] définissent une infrastructure pour effectuer le test exhaustif de JHipster. Ils appliquent ensuite quelques stratégies d'échantillonnage sur l'ensemble des configurations obtenues et analysent les resultats obtenus afin d'établir une comparaison entre ces dernières.

Leurs travaux permettent de conclure que : le test exhaustif de JHipster requiert un effort d'ingénierie considérable, ils montrent également que le test par échantillonnage au moyen de la stratégie Random s'avère être efficace pour la détection d'erreurs. Cependant, pour des échantillons de petites tailles, ils recommandent l'utilisation de PLEDGE car pour des échantillons de petites tailles, cette méthode offre une meilleure éfficacité pour la

detctetion de fautes. Au regard de leurs résultats, Halin *et al.* ont initiés un dialogue avec les principaux développeurs du projet JHipster et formulés des recommandations concernant la stratégie de test utilisée par ces deniers. En effet, le budget de test limité conduisait les développeurs à ne tester que 12 configurations sélectionnées sur la base des fonctionnalités les plus utilisées [1]. De ce fait, ils recommandent l'utilisation de PLEDGE pour le choix des configurations à tester.

Le projet JHipster est marqué par une forte évolution et un grand nombre de versions ceci dû à l'arrivée sans cesse de nouvelles technologies et leurs intégrations au projet afin de satisfaire au mieux les exigences des utilisateurs. Cependant, on aimerait savoir si les résultats obtenus par Halin *et al.* [1] sont généralisables avec l'évolution de JHipster. Autrement dit, quelles sont les variations relatives au coût du test exhaustif des nouvelles versions de JHipster ? Les techniques d'échantillonnage évaluées par Halin *et al.* [1] pour la version 3.6.1 nous donnent elles les mêmes résultats pour des versions suivantes de JHipster ? Pouvons-nous faire les mêmes recommandations que Halin *et al.* [1] aux développeurs quant à la stratégie de test à adopter face à l'évolution de JHipster tout en tenant compte d'un budget de test limité ? Pour répondre à ces questions, nous avons dupliqué les travaux de Halin *et al.* [1] sur la version 4.8.2 de JHipster ceci afin de mesurer l'effort de mise à jour de l'infrastructure de test qu'ils décrivent dans [1] mais également d'évaluer le coût nécessaire en terme de ressource utile pour le test exhaustif et de pouvoir conclure sur les stratégies de test adaptées pour le test par sélection de configuration. Nous suivons donc dans ces travaux les mêmes étapes que Halin *et al.*[1].

La première partie de nos travaux consiste à modéliser la variabilité de JHipster 4.8.2 ensuite de mettre à jour le workflow de test décrit dans [1] afin de générer et tester toutes les configurations possibles. La deuxième partie des travaux consiste à catégoriser les Bugs retrouvés par le test exhaustif et d'appliquer les mêmes stratégies d'échantillonnage que Halin *et al.* [1] afin d'évaluer leurs capacités à détecter des fautes. La dernière partie de nos travaux consiste à présenter les résultats obtenus et à effectuer une discussion comparative avec les résultats de Halin *et al.* [1] afin de définir l'impact de l'évolution de JHipster sur la stratégie de test.

Chapitre 2

État de l'art

Introduction

En génie logiciel, les activités de test et de maintenance représentent parfois 50% du coût de développement. De plus, pour les systèmes à forte variabilité, ces activités tendent à devenir très complexes et coûteuses à cause du nombre élevé de produits générés. Cette section a pour objectif de présenter une revue de littérature sur les activités de maintenance corrective/évolutive et de test dans le cadre de système particulier : les lignes de produits logiciels.

2.1 Les lignes de produit logiciel (LdPL)

Faisant face aux coûts élevés de production de la plupart de produit logiciel, à la qualité pas toujours satisfaisante et aux besoins sans cesse croissants, l'industrie du logiciel a adapté les idées de réutilisation de masse instaurées dans les années 1900 par Henri Ford dans le secteur automobile. Cela permettra non seulement d'augmenter la qualité du logiciel entre deux versions mais également de diminuer l'effort de développement tout en réduisant par la même occasion le coût et les délais de production. La notion de ligne de produit logiciel

(LdPL) est une réponse à ces défis.

2.1.1 Définition et Exemple

Il existe dans la littérature plusieurs définitions de la notion de ligne de produit logiciel. Clement *et al.* [5] définissent une ligne de produits logiciels comme étant un ensemble de systèmes à logiciel prépondérant partageant de manière cohérente un ensemble de caractéristiques communes. Jan Bosh [13] défend qu'elle consiste en l'agrégation d'une architecture, d'une ligne de produit et d'un ensemble de composants réutilisables, conçus pour être incorporés dans l'architecture de la ligne de produit. La ligne de produit étant définie par l'ensemble des produits logiciels développés sur base d'une architecture et des composants. Ces deux définitions bien que différentes ont un point en commun à savoir la notion de composantes réutilisables. Les LdPLs permettent de produire des produits différents qui partagent le même noyau commun de composants tout en permettant des points de variations spécifiques, afin de répondre à des exigences spécifiques à chaque produit. A titre d'exemple de LdPL, on peut mentionner le logiciel qui génère des applications de vente en ligne. Les différents magasins en ligne diffèrent généralement les uns des autres, par exemple dans les méthodes de paiement, les options d'expédition ou les types d'articles pris en charge. Leurs concepts sous-jacents sont très courants et peuvent être mis en œuvre à partir d'actifs réutilisables. Ainsi, une entreprise offrant un service de développement d'applications de vente en ligne peut utiliser des techniques LdP pour obtenir une réutilisation systématique en identifiant et créant les actifs réutilisables requis et en dérivant les produits individuels en fonction des exigences spécifiques du client.

2.1.2 Ingenierie des lignes de produits logiciels

Selon Stephen Creff [3], la philosophie de réutilisation du logiciel sous forme de lignes de produits consiste en la réutilisation intra-organisationelle (composant) réalisée grâce à l'exploitation explicitement planifiée de similitudes entre produits connexes. Pour permettre ce niveau de réutilisabilité, l'ingénierie des lignes de produits logiciels s'organise autour

de deux processus : L'ingénierie du domaine et l'ingénierie d'application. Pohl *et al.* [7] décrivent par la figure 2.1 la distinction entre ces deux processus d'ingénieries.

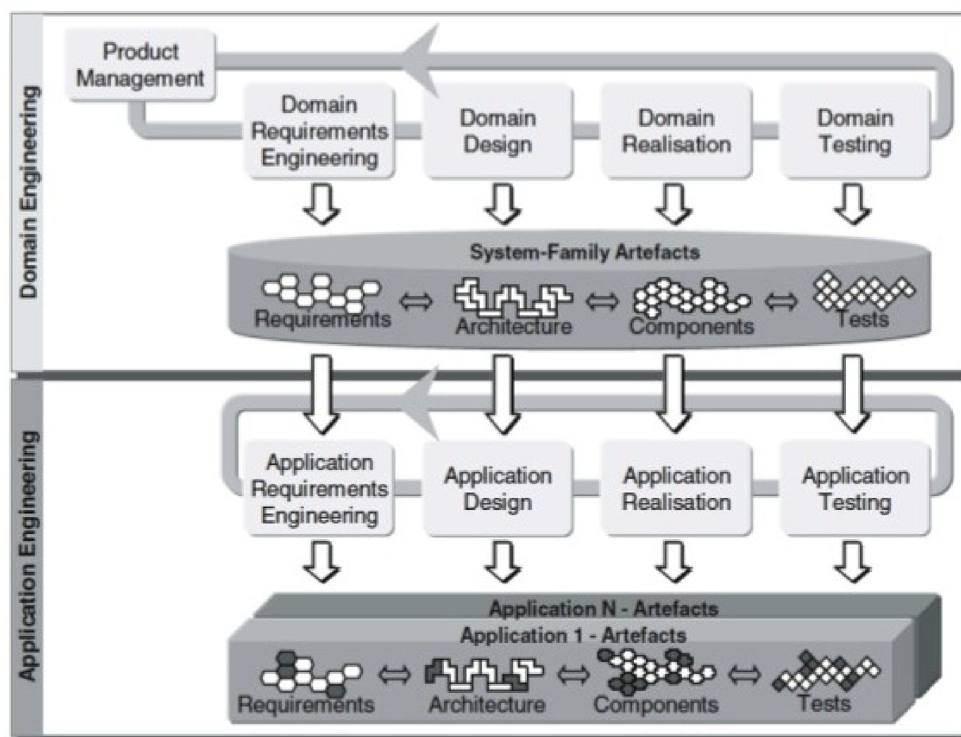

FIGURE 2.1 – Processus d'ingenierie des lignes de produits logiciels [7]

Les processus relatifs à l'ingénierie du domaine permettent de créer et maintenir la ligne de produits logiciels. Les exigences relatives à la LdPL sont toutes d'abord élucidées. Ensuite, la variabilité découlant de la LdPL et relative aux produits potentiels est capturée dans un modèle de variabilité. La dérivation d'un produit spécifique consistera alors à un assemblage de composants réutilisables de la LdPL et d'une combinaison de variantes. Durant les phases de l'ingénierie d'application, des produits concrets sont développés sur la base des actifs fournis par la LdPL. Un produit est défini par une configuration des éléments qui résout la variabilité en sélectionnant les variantes compte tenu des contraintes définies.

2.2 Modélisation de la variabilité et derivation des produits d'une LdPL

2.2.1 Notion de variabilité

Dans les lignes de produits, la variabilité peut être définie comme étant la capacité d'un artefact de base à s'adapter aux différents contextes d'un produit de la LdPL [6]. La variabilité permet donc à un système de répondre à des exigences spécifiques à chaque produit issu d'une LdPL, elle permet ainsi de différencier les produits issus de la LdPL , mais également d'accroître ou de diminuer le nombre de produit (en général, plus il y a de Features dans la LdPL, plus il y a de produits mais avec les contraintes on peut avoir des cas ou une feature réduit le nombre de produits). Des travaux récents permettent de classer la variabilité en deux catégories. Pohl *et al.* [4] définissent deux types de variabilités : La variabilité externe représentant celle visible par le client, par exemple, Pour le cas des smart phone décrit précédemment, nous avons comme variabilité externe une caméra, la géolocalisation ou encore reconnaissance vocale et la variabilité interne se rapprochant quant à elle des éléments du domaine et n'est pas tout à fait visible. Pohl *et al.* [4] définissent également la variabilité sur le plan temporel et sur celui de l'espace en montrant que la variabilité temporelle peut être une émanation de la maintenance corrective/évolutive du système ou à la gestion des différentes configurations ou versions.

2.2.2 Modélisation de la variabilité d'une LdPL

Il existe plusieurs modèles permettant de représenter la variabilité dans une LdPL (modèle de variabilité), on peut par exemple citer le modèle OVM, encore appelé modèle de variabilité orthogonale qui s'articule autour de la notion de point de variation [3] et modélise les éléments variables de la LdPL. On peut également citer le Feature Model ou modèle caractéristique, ce dernier modèle est celui utilisé dans ces travaux et donc celui que nous décrivons dans le document.

2.2.2.1 Le Feature Model

Kang *et al.* [8] ont introduit la notion de Feature Model ou modèle caractéristique en 1990 pour décrire la variabilité d'une LdPL dans le cadre de leur méthode FODA (Feature-oriented Domainn Analysis). La modélisation de la variabilité par les Features est selon Acher *et al.* [9] l'une des techniques les plus utilisées pour modéliser la variabilité dans une LdPL et raisonner sur un tel système. Les Features peuvent être utilisées pour distinguer les produits de la LdP. Kang *et al.* [8] se sont basés sur cette notion pour proposer la méthode FODA. L'idée de FODA est d'utiliser la notion de Feature pour organiser les exigences d'un ensemble de produits similaires dans une LdPL ceci afin de faciliter la spécialisation de ces exigences pour des produits spécifiques. On utilise donc un arbre ou chaque nœud représente une Feature et il existe des relations et des contraintes entre elles : c'est le «Feature Model» ou modèle à caractéristique. Ces modèles utilisent cinq notions pour décrire les relations entre les Features :

- la dépendance obligatoire «Mandatory» : exprime le fait que lors de la dérivation de chaque configuration ou produit, la sous-Feature du parent déclarée « mandatory » doit être sélectionnée.
- la dépendance optionnelle «Optional» : exprime le fait que lors de la dérivation de chaque configuration ou produit, la Feature définie comme optionnelle peut être sélectionnée ou non.
- la dépendance à choix alternatif «Alternative ou XOR» : exprime le fait que lors de la dérivation de chaque configuration ou produit, parmi les sous-Features du parent, une seule doit être sélectionnée.
- la dépendance conjonctive «AND» : exprime le fait que lors de la dérivation de chaque configuration ou produit, toutes les sous-Features du parent, doivent être sélectionnées.
- la dépendance disjonctive non exclusive «OR» : exprime le fait que lors de la dérivation de chaque configuration ou produit, une ou plusieurs sous-Features du parent doit être sélectionnées.

On distingue pour ce modèle deux types de contraintes : « Requires » et « Excludes ». la

figure 2.2 représente le Feature Model caractérisant l'exemple de l'e-shop. De nombreux

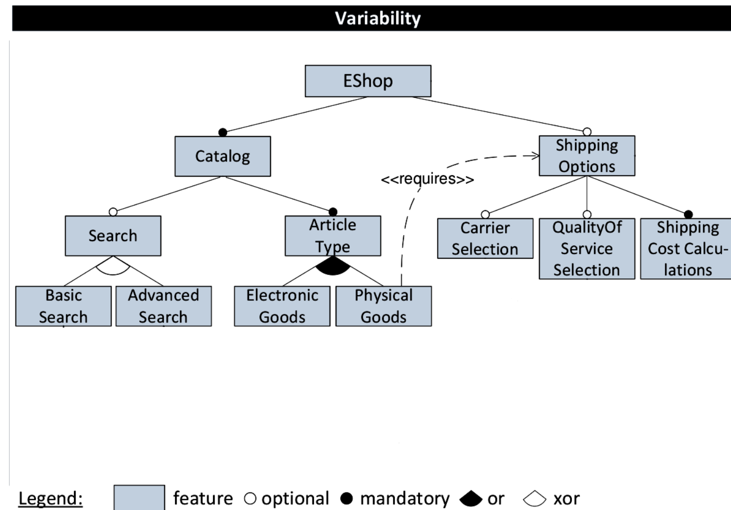

Figure 2.2 – Exemple de Feature Model de decrivant l'e-shop

travaux de recherche ont permis d'étendre le Feature Model en introduisant de nouvelles informations telles que la cardinalité des Features ou encore les attributs. A titre d'exemple, Czarnecki *et al.* [10] proposent un modèle de Features avec des cardinalités. Ce modèle représente une extension de FODA dans laquelle à chaque Feature est associée des attributs et une cardinalité représentant le nombre d'instances de la Feature pouvant être utilisée dans une configuration. Le modèle présenté par Czarnecki *et al.* propose également l'utilisation du langage OCL pour exprimer les contraintes entre différentes Features porteuses de cardinalité.

2.2.3 La dérivation des produits

La dérivation de produit dans une LdPL consiste en génération d'un ou de plusieurs produits spécifiques. Cela revient concrètement à réaliser des produits sur base d'exigences et par spécialisation des artefacts partagés par l'ensemble des produits de la LdPL. Il existe dans la littérature plusieurs approches de dérivation des produits. Ces approches peuvent se classer en deux grandes catégories : la dérivation par configuration et la dérivation par transformation [3].

2.2.3.1 Dérivation par configuration

Elle est basée sur la paramétrisation. Le produit est généré en fournissant une ou plusieurs exigences et paramètres spécifiques. Les activités de dérivation se découpent en deux étapes : La configuration qui consiste en une série de choix définissant les exigences et les contraintes, suivie de la réalisation qui consiste à générer le produit sur base du résultat de la configuration. Un exemple illustratif est défini dans les travaux de Botterweck *et al.* [17], les auteurs présentent une approche de dérivation des produits par configuration. La première phase consiste en la modélisation et l'implémentation de la LdPL au moyen de Feature Diagram, puis la dérivation des configurations spécifiques via des exigences et des contraintes spécifiques. La seconde phase consiste à dériver un produit exécutable au moyen de procédures implémentées dans un langage de programmation. Dans ces travaux, nous utiliserons ce type de dérivation.

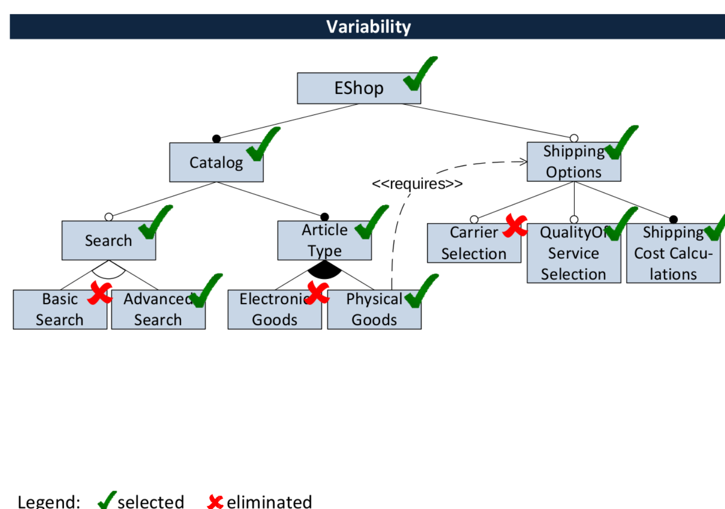

FIGURE 2.3 – Exemple de configuration de l'e-shop [12]

2.3 Évolution des LdPLs

Selon la première loi de Lehman sur l'évolution des logiciels, un logiciel doit être continuellement adapté ou il deviendra de moins en moins satisfaisant. Ceci voudrait tout simplement

dire que pour ne pas devenir obsolète, tout logiciel est appelé à évoluer au cours du temps. Les lignes de produits logiciels n'échappent pas à cette loi. L'évolution des LdPLs peut être observée suivant deux vues : la vue organisationnelle(architecture) et la vue des processus [11]. Cette évolution est dirigée par des changements de plusieurs natures : Nouvelles exigences, modification des exigences existantes sur un ou plusieurs produits de la LdPLs. L'apparition de tous ces changements provient généralement de sources diverses(besoins utilisateurs, besoins futurs d'une entreprise, introduction de nouvelles fonctionnalités ou de nouveau produit dans la LdPL etc.). L'évolution des LdPLs fait face à plusieurs challenges causés par leurs caractéristiques [12] :

Longue durée de vie : d'une part, une LdPL est un investissement à long terme et n'est productif que dans la mesure où elle dérive toujours plus de produits. D'autre part, la LdPL doit évoluer et modifiées de ses produits afin refléter les nouvelles exigences. Par conséquent, une LdPL évoluera souvent et sur une longue période de temps afin de garantir sa rentabilité et également adapter ses produits aux nouvelles exigences.

Grande Taille et complexité : la LdPL représente toute une famille de produits, elle est de grande taille et donc plus complexe qu'un produit individuel. Généralement, plusieurs équipes sont impliquées dans sa création et sa maintenance. Par conséquent, les connaissances peuvent être plus distribuées et l'évolution des différentes parties d'une Ldp peut se produire à des vitesses différentes.

Interdépendance entre modules : en raison de la réutilisation systématique des composants d'une LdPL, il existe plus d'interdépendances entre les ressources logicielles. Les modifications dans la LdPL (Ex. Une correction de Bug dans un actif réutilisable) peuvent affecter plusieurs produits individuels créés en fonction de la LdPL et les nouvelles exigences au niveau du produit peuvent nécessiter des modifications de toute la LdPL.

2.3.1 Une approche d'évolution des lignes de produits logiciels

L'évolution d'un système peut impliquer des changements qui sont dûs soit à la création des nouvelles exigences nécessaires pour s'intégrer à la nouvelle architecture soit à des changements liés à la maintenance. S'il n'existe pas de processus pour gérer cette évolution, les changements observés peuvent provoquer des problèmes de conception du système ou des discordances entre les exigences définies et le produit [11]. Afin de maintenir un système logiciel utilisable face à ces changements , il devient important de définir des processus d'évolution en prévoyant les types de changements qui peuvent survenir dans le temps de façon à ce que l'architecture supporte ces différents changements. Selon Mens *et al.* [12], il existe deux types de problèmes majeurs lorsqu'on essaye de supporter l'évolution d'une LdPL. Le premier concerne les difficultés à minimiser le nombre de changements nécessaires à l'évolution et le second concerne la minimisation de la complexité nécessaire pour gérer l'évolution. Il existe dans la littérature plusieurs processus et méthodes pour l'évolution des LdPLs Mens *et al.* [12] suggèrent l'utilisation des Features Models comme moyen d'abstraction pour décrire l'évolution globale d'une LdPL car ils décrivent aux mieux les exigences des parties prenantes. L'évolution d'une LdPL est donc représentée comme une séquence de Features Models dans le temps. Les auteurs prennent comme exemple le domaine de l'e-shop décrit par la figure 2.4 qui présente quatre versions de Feature Model de la LdPL à quatre moments différents, y compris l'évolution historique (2012), présente (2013) et les étapes d'évolution futures prévues (2014 et 2015). En 2012, la version ne prend en charge qu'un catalogue et les options d'expédition avec la caractéristique **"CarrierSelection"** qui est optionnelle. La version de 2013 (courante) a été prolongée par ajout de la recherche qui est disponible soit en tant que **"BasicSearch"** ou **"AdvancedSearch"**. La version 2014 fera la distinction entre **"ElectronicGoods"** (qui peut être expédié ou téléchargé directement) et **"PhysicalGoods"** qui doivent être expédiés. Par conséquent, Shipping-Options est devenue une fonctionnalité facultative et une contrainte a été ajoutée. Dans cette version, **"AdvancedSearch"** ne sera pas supporté car il nécessite plus de temps pour l'intégrer avec les changements sur le catalogue. Pour 2015, il est prévu de soutenir un type d'article supplémentaire Services et de soutenir **"AdvancedSearch"** à nouveau pour tous les types d'articles. Il existe deux méthodes pour spécifier les changements entre différentes

versions d'un artefact en utilisant des Features Models [12] : la première consiste à spécifier les différences entre les modèles et la seconde consiste à décrire au moyen d'opérateur, les modifications effectuées sur les modèles.

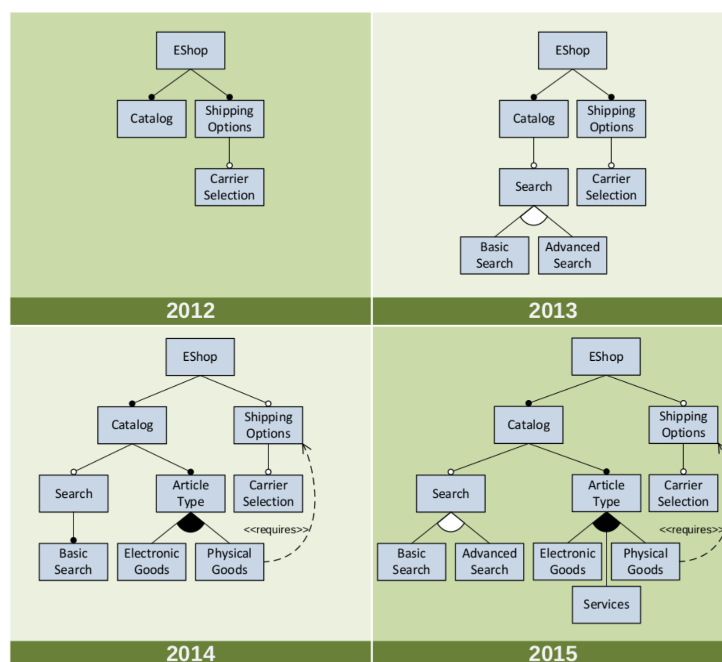

FIGURE 2.4 – Evolution d'une LdPL par Feature Model [12]

2.3.1.1 Approche d'évolution par différences de Modèles

Cette approche d'évolution est similaire aux approches de différenciations de programmes[12]. En effet, elle spécifie les différences entre des versions de feature modèles. On construit alors un modèle de différence contenant les changements entre deux versions en termes d'éléments ajoutés, supprimés et modifiés. La figure 2.5 montre un exemple d'évolution de 2013 à 2014 dans l'exemple de l'e-shop : le groupe xor et son noeud **"AdvancedSearch"** ont été supprimés. Les caractéristiques ArticleType, **"ElectronicGoods"** et **"PhysicalGoods"** ainsi que leurs relations et contraintes ont été ajoutées. En outre, **"ShippingOptions"** a été modifié pour devenir une fonctionnalité facultative. Les éléments de contexte (représentés par la couleur de la lumière dans la Figure 9.10) sont utilisés pour spécifier les

emplacements dans le modèle.

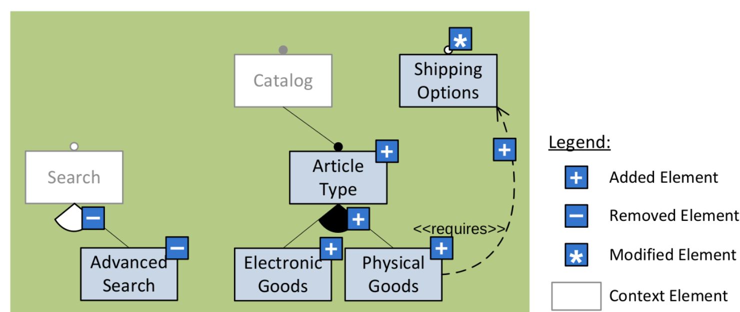

FIGURE 2.5 – Différence de Modèle pour l'évolution [12]

2.3.1.2 Approche d'évolution par opérateur de changement

Le deuxième concept de base pour spécifier les changements sont les opérateurs de changement. Un opérateur de changement décrit une opération effectuée sur un modèle pour réaliser un changement. Ce modèle utilise trois opérateurs de modification atomique : l'ajout, la suppression et la modification ; ces opérateurs ont la même sémantique que ceux décrits pour les modèles de différences. Cependant, la principale différence étant la possibilité d'utiliser des opérateurs plus complexes qui permettent d'exprimer une sémantique plus riche à propos d'un changement comme par exemple "diviser le Feature A en deux nouvelles Features A1 et A2".

2.4 Test des produits d'une lignes de produits logiciels

L'activité de test des lignes de produits logiciels pourrait consister à tester la validité de tous les produits dérivés de cette dernière. L'effort de test serait alors proportionnel au nombre de variantes ou de configurations susceptibles d'être générées. Cependant, il est généralement assez difficile de générer tous les produits d'une LdPL, ceci à cause de leur nombre élevé dû à une variabilité élevée (le nombre de fonctionnalités élevé conduit à des

milliers de produits différents). Tous ces points rendent l'activité de test des LdPLs assez complexe et coûteuse, car nécessite plus de ressources et un effort d'ingénierie conséquent. Cette section présente quelques techniques tirées de la littérature permettant de tester une LdPL.

2.4.1 Approche en force

La stratégie de test en force consiste à effectuer toutes les activités de test pour tous les produits possibles au cours de l'ingénierie de domaine. Cela consisterait à dériver à partir d'un Feature Model représentatif de la LdPL, tous les produits valides et de tester chaque produit obtenu. Comme montré par Halin *et al.* [1] cette stratégie est coûteuse, non applicable dans la plupart des cas, d'où l'intérêt pour des techniques visant à réduire l'espace des produits à tester.

2.4.2 Approche par sélection des cas de tests

Étant donné le nombre exponentiel de produits pouvant être obtenus d'une LdPL, il est difficile de tous les tester. Une solution est donc la sélection des cas de tests. Les techniques de selection des cas de tests prennent pour postulat le fait que les bugs sont très souvent le résultat de l'interaction entre certaines Features. Elles essayent alors de sélectionner un ensemble de produits couvrant le plus grand nombre de combinaisons de Features possibles. Cela permet de tester moins de produits et ainsi réduire le coût du test. La figure 2.6 présente la stratégie de sélection de cas de test. Il existe plusieurs variantes

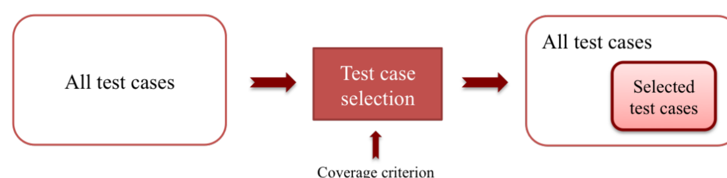

FIGURE 2.6 – Selection des cas de tests

de cette technique. On peut entre autre citer : l'approche de test par critère de sélection

pairwise (généralisée à T-wise) de Lochau *et al.* [21]. On peut également citer l'approche de sélection par critère de dissimilarité et l'approche de sélection aléatoire ou Random. Dans le cadre de ces travaux nous utiliserons ces différentes stratégies de sélection afin de réduire l'ensemble des configurations à tester.

2.4.2.1 Sélection par interaction combinatoire

Le test d'interaction combinatoire est basé sur l'observation selon laquelle, la plupart des défauts relatifs aux produits d'une LdPL sont causés par l'interaction entre un petit nombre de variables. En effet, une interaction entre Features se produit lorsqu'une ou plusieurs Features modifient ou influencent le comportement d'autres Features. Les techniques de test d'interaction combinatoire sont utilisées pour réduire le nombre de produits à tester en sélectionnant un sous ensemble de produits ou de configurations valides satisfaisant la couverture. Ces techniques fonctionnent généralement en définissant d'abord un modèle de l'espace de configuration du système (un Feature Model par exemple). Généralement, le modèle des interactions de features dans la LdPL comprend un ensemble d'options de configurations, chacune pouvant prendre un petit nombre de paramètres d'options et un ensemble de contraintes inter-options à l'échelle du système (toutes les configurations ne sont pas forcement valides). Compte tenu de ce modèle, les méthodes de test par interaction combinatoire (Combinatorial interaction testing) calculent ensuite un petit ensemble de configurations concrètes, une matrice de recouvrement, dans laqelle chaque configuration valide contenant une combinaison de t (nombre de features en interaction) options apparaît au moins une fois. Enfin, le système est testé en exécutant sa suite de tests sur chaque configuration de la matrice de couverture. Cette idée peut être réutilisée pour le cas des Features. Les tests combinatoires peuvent être alors utilisés pour couvrir des interactions de type T-wise entre Features. Cohen *et al.* [28] utilisent l'OVM pour modéliser les parties variables et communes de la LdPL, qui sont décrites par un modèle relationnel. Ce modèle relationnel sert de base sémantique pour définir les critères de couverture pour la LdPL testé. Dans [29], les auteurs utilisent l'approche de Cohen *et al.* [28] pour sélectionner les produits à tester. Perrouin *et al.* [20] utilisent une approche similaire à celle de Cohen *et*

al. [28]. La différence significative est que Perrouin *et al.* [20] utilisent un solveur SAT et non un modèle relationnel. De manière plus générale, la technique T-wise sélectionne l'ensemble de toutes les combinaisons telles que tous les T-uplets possibles de valeurs des Features soient incluses dans l'ensemble des données de test. Les tests T-wise échantillonnent l'ensemble des données en entrée pour couvrir toutes les combinaisons de T Features. Dans le contexte du test d'une LdPL, cela consiste à sélectionner l'ensemble minimal de produits dans lequel toutes les interactions T-wise entre les Features se produisent au moins une fois. Exemple : soit le Feature Model de la figure 2.7. Les combinaisons pairwise (T=2) valides

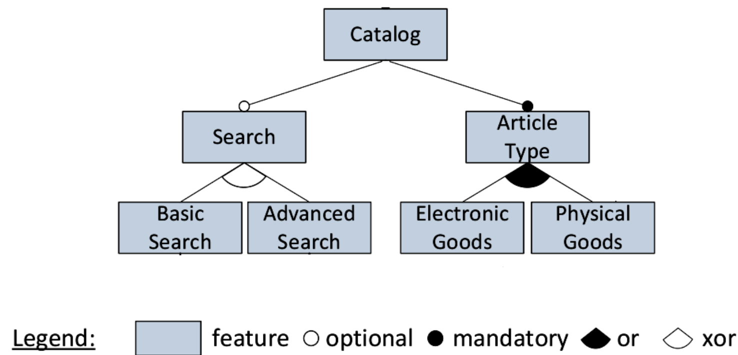

FIGURE 2.7 – Description de pairwise

de la caractéristique feature Basic search (B) et de la caractéristique Advanced search(A) du modèle de la figure 2.7 sont : $B \wedge \neg A$ et $\neg B \wedge A$. Les combinaisons non valides sont $\neg B \wedge \neg A$ et $B \wedge A$. Chaque combinaison valide doit apparaître au moins dans l'une des configurations créées de l'échantillon. La création de ces différentes configurations est une instances du problème de la matrice de recouvrement [22]. En particulier, les configurations peuvent être représentées sous forme de tableau de recouvrement. Le principal défi de la génération des tableaux de recouvrement consiste à trouver le nombre minimal de configurations couvrant les pairwise combinaisons de caractéristique. De nombreux algorithmes ont été proposés pour se rapprocher de ces tableaux de recouvrement minimaux on peut par exemple citer **ICPL** décrit par Johansen *et al.* [23] permettant une couverture T-wise avec ($T \in 1,...,3$) ; **Chvatal** défini par Chvatal [24] permettant une couverture T-wise avec ($T \in 1,...,4$) et **IPOG** de Lei *et al.* [25] permettant une couverture T-wise avec ($T \in 1,...,6$). Des outils implémentant ces algorithmes ont également été developpés afin de pouvoir générer des ensembles vérifiant les critères de couverture définis, c'est le cas

de **PACOGEN** défini par Aymeric Hervieu *et al.* [26], **MoSo-PoLiTe** défini par Oster *et al.* [27] et **SPLCATool** de Johansen *et al.* [23] qui pour ce dernier implémente la plus part des algorithmes cités précédemment. Dans la suite de nos travaux, nous appliquerons la technique T-wise eu moyen de l'outil SPLCATool afin d'éffectuer l'échantillonnage des produits à tester.

2.4.2.2 Échantillonage basé sur la dissimilarité

L'échantillonnage des produits basé sur la dissimilarité repose sur l'idée selon laquelle les produits non similaires sont plus susceptibles de détecter les bugs que les produits similaires. Il est donc question d'échantillonner un ensemble de produits aussi différent que possible, basé sur une mesure de distance. Dans leurs travaux, Henard *et al.* [30] considèrent la distance entre les produits en termes de Features qui diffèrent. Deux produits sont différents si les Features qui les composent sont différentes. L'échantillonnage est effectué en utilisant un algorithme évolutionnaire qui prend une taille fixe de l'échantillon et une durée de temps comme paramètres. La distance de Jaccard [31] entre deux produits est utilisée pour calculer la valeur de réajustement de l'échantillon à chaque itération. Lors de l'exécution, les produits sont classés dans l'échantillon en fonction de leur dissemblance. Finalement, l'algorithme produit une liste classée de produits avec éventuellement les produits les plus dissemblables en premières positions. Contrairement à l'algorithme T-wise, la taille de l'échantillon est fixé depuis le début. Cela est très pratique lorsque le budget de test est limité à un nombre donné de produits. L'algorithme d'échantillonnage basé sur la dissimilarité est implémenté dans PLEDGE [30]. Dans la suite nous utiliserons donc PLEDGE pour définir cette stratégie d'échantillonnage.

2.4.2.3 Échantillonage aléatoire(Random sampling)

L'échantillonnage aléatoire (Random sampling) fait partie de la technique d'échantillonnage dans laquelle chaque produit a la même probabilité d'être choisi. L'échantillon est construit en choisissant chaque produit au hasard et l'échantillon ainsi construit est censé

être une représentation impartiale de la population totale. Si pour certaines raisons, l'échantillon ne représente pas la population, la variation est appelée erreur d'échantillonnage. L'échantillonnage aléatoire est l'une des formes les plus simples de collecte de données sur la population totale. Sous l'échantillonnage aléatoire, chaque produit de l'échantillon construit a la même probabilité d'être choisi. Cette stratégie requiert la donnée de la taille de l'échantillon à obtenir en entrée du processus d'échantillonnage.

2.4.2.4 Échantillonage par activation/désactivation de Feature

Décrite par Abal *et al.* [32], cette méthode consiste à créer un échantillon de configuration à tester par activation ou désactivation d'un ou de plusieurs Features d'un système hautement configurable. Il existe dans la littérature plusieurs algorithmes de sélection suivant cette logique. On peut par exemple citer : One-Disabled [32], One-Enabled et Most-enabled-disabled.

One-Disabled
C'est un algorithme proposé par Abal *et al.* [32] basé sur une observation de 42 failles trouvées dans le noyau linux. En effet Abal *et al.* remarquent qu'en sélectionnant un ensemble de configuration valide en désactivant pour toute configuration les options de configurations une à la fois il obtenait un ensemble assez représentatif des configurations provoquant les 42 failles observées. A titre d'exemple, soit le Feature Model de la figure2.8, ce Feature Model permet d'obtenir 3 configurations valides à savoir : les configurations ***AB*, *ABCF* et *ABCG***. cependant l'algorithme One-Disabled sélectionnera un sous ensemble de configuration valide en désactivant chaque Feature. Ainsi il sélectionnera parmi : $\neg ABCDE$, $A\neg BCDE$, $AB\neg CDE$, $ABC\neg DE$, $ABCD\neg E$. Parmi ces configurations obtenus, seule deux sont valides : ***ABC¬DE* et *ABCD¬E***. L'échantillon obtenu est donc dans ce cas l'ensemble ***Ens*={ *ABC¬DE* ,*ABCD¬E*}.**

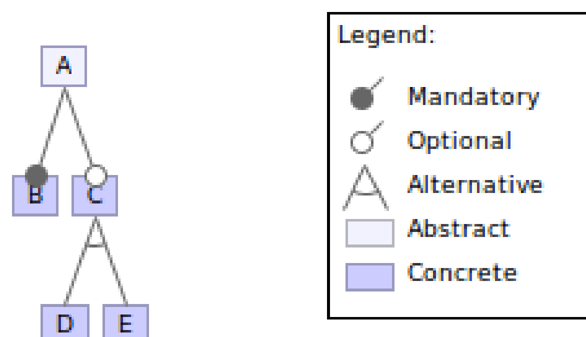

FIGURE 2.8 – Feature Model illustratif de la technique One-Disabled

One-Enabled

cet algorithme quant à lui est basé sur le même principe que le précédant sauf que ici on active les Features les uns après les autres. Son exécution sur le Feature Model de la figure2.8 nous donnerait comme choix les configurations : $A\neg B\neg C\neg D\neg E$, $\neg AB\neg C\neg D\neg E$, $\neg A\neg BC\neg D\neg E$, $\neg A\neg B\neg CD\neg E$, $\neg A\neg B\neg C\neg DE$. Aucune de ces configurations n'est valide on conclut que l'échantillon sera vide (cette methode ne permet pas de construire un échantillon avec les configurations disponibles pour ce Feature Model).

Most-enabled-disabled

L'algorithme Most-enabled-disabled publié par Medeiros *et al.*[33] quant à lui sélectionne les configurations dans lesquelles la plupart des Features optionnelles sont activées et les configurations dans lesquelles la plupart des Features optionnelles sont désactivées. Pour l'exemple de la figure2.8, les configurations sélectionnées seront :

— **La Feature optionnelles (*C* est activée)** : on obtient les configurations $ABCD\neg E$, $ABC\neg DE$

— **La feature optionelle (*C* est désactivée)** : on obtient la configuration AB

L'échantillon obtenu dans ce cas est donc l'ensemble $\boldsymbol{Ens}=\{\boldsymbol{AB}, \boldsymbol{ABCD\neg E}, \boldsymbol{ABC\neg DE}\}$

2.4.3 Approche par utilisation des cas de test

Dans le cadre des LdPLs, en raison du grand nombre de produits potentiels pouvant être dérivés et des contraintes de coût et de temps, il est difficile d'exécuter tous les cas de test dans une suite de tests existante. Pour diminuer le coût des tests, différentes techniques ont été proposées dans la littérature et la plupart d'elles proposent de réduire l'ensemble de cas de test suivant des critères bien définis. Parmi ces techniques, nous en présenterons dans la section suivante trois : la priorisation, la minimisation et la sélection.

2.4.3.1 Minimisation des suites de test

Le but de la minimisation des suites de test c'est de générer un ensemble représentatif de la suite de tests qui satisfassent toutes les mêmes exigences que la suite de tests d'origine et avec un nombre minimum de cas de tests. L'objectif principal des techniques de minimisation des tests élémentaires est de supprimer les cas de test qui deviennent redondants et obsolètes avec le temps [19]. Plusieurs techniques ont été proposées dans la littérature et peuvent être classées en trois catégories : les techniques heuristiques, les algorithmes génétiques, les techniques basées sur la programmation linéaire.

2.4.3.2 Priorisation des cas de test

Cette technique permet de déterminer un ordonnancement efficace des cas de test afin de faciliter la détection des Failures. La plus part des travaux relatifs au test par priorisation sont basés sur des tests d'interaction combinatoire. En effet, elles génèrent des configurations combinatoires en utilisant des poids pour les hiérarchiser. Hernard *et al.* [30] ont présenté une approche de priorisation basée sur des similitudes pour générer des cas de test de la gamme de produits logiciels tout en maximisant la couverture en T-wise. Cette approche combine l'échantillonnage et la priorisation pendant la génération du test. Certains autres travaux n'utilisent pas les tests d'interaction combinatoire pour l'échantillonnage de leurs cas de test. Devroey *et al.* proposent une approche de test basée sur l'usage d'un

modèle pour prioriser les tests de la LdPL [34]. Cette approche repose sur un modèle de variabilité, un système de transition de Feature et un modèle définissant les probabilités d'exécuter des transitions dérivées des usages du système en analysant ses logs d'exécution. Cette approche combine des concepts issus de tests statistiques et l'échantillonnage de la LdPL pour extraire des produits en fonction de la probabilité de leurs traces d'exécution.

Chapitre 3

Cas d'étude : JHipster

Notre étude s'est portée sur le générateur d'application Web JHipster. Il s'agit en effet d'étudier l'évolution de JHipster et d'évaluer son impact sur l'effort et la stratégie de test définie par Halin *et al* [1]. Cette section a pour but de décrire le générateur d'application JHipster sur deux points (le test et l'évolution). Elle décrit également les motivations ayant guidées le choix de notre problématique.

3.1 Test de JHipster

3.1.1 JHipster

JHipster [1] est une application open source qui associe les technologies Spring Boot côté serveur et AngularJs/Bootstrap côté client pour générer des applications web[14] . Projet débuté en 2013 par Julien Dubois, JHipster supporte aujourd'hui un grand nombre de technologies. En effet, il prend en entrée une combinaison d'outils, de Framework et une architecture d'application afin de générer une application totalement fonctionnelle . La génération d'une application par JHipster se déroule en deux phases : la configuration et

1. http ://www.jhipster.com/

la génération [14]. La configuration par l'utilisateur consiste via la ligne de commande à donner en input les technologies, les outils et l'architecture de l'application à générer. Le résultat de cette phase est la génération d'un fichier yo-rc.json décrivant la configuration. La figure C.7 décrit l'interface de configuration de JHispter.

```
koko@koko-Lenovo-G50-70:~$ jhipster
Using JHipster version installed globally
Running default command
Executing jhipster:app
Options:

JHIPSTER

http://www.jhipster.tech

Welcome to the JHipster Generator v4.8.2
Documentation for creating an application: http://www.jhipster.tech/creating-an-app/
Application files will be generated in folder: /home/koko
 ____________________________________________________________________________

  JHipster update available: 5.1.0 (current: 4.8.2)

  Run yarn global upgrade generator-jhipster to update.

 ____________________________________________________________________________

? (1/16) Which *type* of application would you like to create? (Use arrow keys)
❯ Monolithic application (recommended for simple projects)
  Microservice application
  Microservice gateway
  [BETA] JHipster UAA server (for microservice OAuth2 authentication)
```

FIGURE 3.1 – Interface de configuration de JHipster

La génération de l'application consiste à la création d'une application fonctionnelle sur base des spécifications décrites dans le fichier yo-rc.json. JHipster est donc un exemple de système configurable : les applications générées sont obtenues par une combinaison des variables prises en entrée fournis au générateur JHipster (Technologies, outils, l'architecture, etc.) .

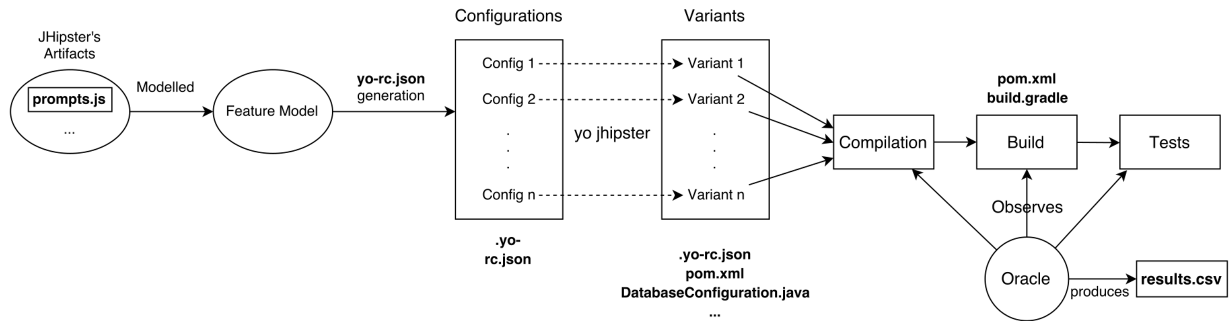

FIGURE 3.2 – Processus de test des configurations de JHipster [1]

3.1.2 Tester JHipster

Dans le but d'apporter des réponses à la problématique du test des systèmes configurables, Halin *et al.* [1], évaluent le coût de test d'un système configurable particulier : JHipster. Ils évaluent également le gain apporté par l'utilisation d'une méthode de test par sélection : Le sampling pour tester les configurations générées par JHipster. Ils décrivent concrètement la conception et l'implémentation d'une suite de procédure permettant de générer toutes les configurations de JHipster 3.6.1 et de les tester (test exhaustif) . Ils évaluent ensuite l'efficacité de quelques méthodes de sampling pour le choix des configurations à tester à savoir T-wise[21] implémenté au moyen de SPLCAT tool [23], Sampling par Dissimilarité[30] implémenté au moyen de PLEDGE[30], le Random sampling ainsi que les techniques Most-enabled-disabled,One-disabled et One-enabled [32]. Dans leurs travaux, Halin *et al.* [1] commencent par implémenter une infrastructure pour tester de manière exhaustive les différentes configurations de JHipter. Ils découpent alors le processus de test en plusieurs activités : L'extraction de la variabilité et la dérivation des configurations, la génération des applications, la compilation suivie de la construction des différentes applications correspondant aux configurations générées. La figure 3.2 décrit ces différentes étapes.

Une fois le processus appliqué à toutes les configurations obtenues, Halin *et al.* [1] effectuent une analyse des résultats afin de répondre aux questions de savoir :

- Quel est l'effort d'ingénierie nécessaire pour tester toutes les configurations ?

- Quelles sont les ressources de calcul nécessaires pour tester toutes les configurations ?
- Quels sont les types de Bugs et dans quelles proportions sont ils rencontrés ?
- Parmi les méthodes d'échantillonnage utilisées, laquelle est la plus efficace pour la sélection des configurations à tester (détecte le plus de Bug représentatif) ?

3.1.2.1 Extraction de la variabilité et dérivation des configurations

Durant cette phase, Halin *et al.* [1] ont tout d'abords extrait la variabilité de JHipster en le décrivant via un modèle de variabilité. La modélisation s'est faite en considérant les différentes fonctionnalités comme des attributs porteurs de variabilité et en tenant également compte des contraintes sur ces fonctionnalités. Ces contraintes et attributs sont explicitement décrites dans le fichier Client/prompts.js et Server/prompts.js du générateur JHipster. Le modèle obtenu a ensuite été implémenté au moyen du langage *Familliar* [2] et a permis de générer toutes les configurations, possible de JHipster 3.6.1 encapsulé chacune dans un fichier yorc-json. La figure 3.3 représente un exemple de configuration.

3.1.2.2 Génération,Compilation et Exécution des configurations

Cette étape des travaux de Halin *et al.* [1] a consisté tout d'abord à générer au moyen de la commande yo jhipster l'application correspondant au fichier yorc-json. Ensuite d'utiliser un ensemble de procédures nommées **Oracle** dans la figure 3.2 pour compiler et exécuter les tests. Les résultats produits par cette procédure ont été écrit dans un fichier CSV. Le fichier CSV contient donc pour chaque configuration le résultat de la phase de compilation, d'exécution et de test. Ce découpage a permis l'implémentation d'un processus regroupant chacune de ces différentes phases. Le résultat de l'implémentation est décrit à l'adresse de ce lien (résultat [3]) et la figure 3.4 repris de [1] décrit les étapes exécutées par le processus implémenté .

2. http ://familiar-project.github.io/
3. https ://github.com/axel-halin/Thesis-JHipster

```
{
 "generator-jhipster": {
  "jhipsterVersion":  "4.8.2",
  "baseName":   "jhipster",
  "packageName":  "io.variability.jhipster" ,
  "packageFolder":  "io/variability/jhipster",
  "serverPort":  "8080",
  "authenticationType": "jwt" ,
  "hibernateCache": "infinispan" ,
  "clusteredHttpSession": false ,
  "websocket": " infinispan"  ,
  "databaseType":  "sql" ,
  "devDatabaseType": "h2Disk",
  "prodDatabaseType":  "mssql",
  "searchEngine":   "elasticsearch" ,
  "messageBroker":  "elasticsearch" ,
  "serviceDiscoveryType":    "elasticsearch",
  "buildTool":    "maven",
  "enableSocialSignIn":       true ,
  "enableSwaggerCodegen": true ,
  "jwtSecretKey":  "c89e91a53523be0e1ae147a50fc31dc06082e719" ,
  "enableTranslation": false  ,
  "applicationType":  "monolith"  ,
 "clientFramework":  "angularX" ,
 "useSass":     true,
  "testFrameworks":  [" gatling " ," cucumber " ," protractor "]  ,
  "jhiPrefix":   "jhi"  ,
  "skipClient":  true ,
  "skipUserManagement":    true ,
  "clientPackageManager":  "yarn"
 }
}
```


Figure 3.3 – Exemple de configuration décrit par le fichier .yo-rc.json

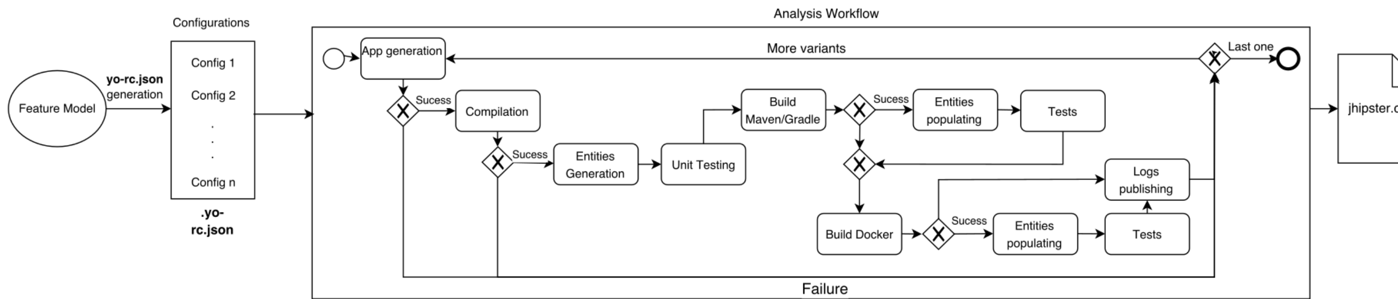

FIGURE 3.4 – Workflow de test de JHipster [1]

3.1.2.3 Exécution du Workflow

Étant donné le nombre élevé de configurations obtenues (26.256) pour la version 3.6.1 étudiée par Halin *et al.* [14], l'exécution du workflow sur toutes les configurations requiert un temps énorme et demande également la mobilisation de beaucoup de ressources. Halin *et al.* [1] ont exécuté le workflow implémenté sur une grille de calcul (GRID 5000), cela leurs a permis d'avoir plus de ressources à leur disposition et de gagner en temps quant à l'obtention des résultats. Les résultats obtenus les ont permis de conclure quant à la quantité de ressource humaine et machine nécessaire pour tester toutes les configurations de JHipster. Ceci leurs permettra de répondre aux deux premières questions. Une fois les résultats des exécutions obtenues, s'en est suivi une analyse quantitative, qualitative et statistique relative aux Bugs obtenus. Ceci permettant aux auteurs de répondre aux deux dernières questions.

3.1.2.4 Les résultats obtenus

En effet, Les travaux de Halin *et al.* [1] arrivent aux conclusions suivantes :

- L'infrastructure de test est en elle-même un système configurable et nécessite un effort d'ingénierie important (8 homme-mois) pour couvrir toutes les activités de conception, de mise en œuvre et de validation.
- Malgré certaines optimisations (par exemple, le pré-chargement en cache des dépendances), le test de toutes les configurations nécessite une quantité importante

de ressources de calcul (4376 heure-machine et 5,2 To d'espace disque) .

- Le test exhaustif de JHipster montre que 34,37% des configurations ne compilent pas ou ne Build pas et 6 Fautes causées par l'interaction de 2 ou 4 Features sont la cause première de ce pourcentage élevé d'échecs ou Erreurs.
- Les techniques one-enabled, one-disabled et most-enabled-disabled identifient moins de défauts que la technique Random pour des échantillons la même taille. La stratégie d'échantillonnage PLEDGE est supérieure à Random pour les échantillons de petite taille. Les stratégies 2-wise ou 3-wise sont légèrement plus efficaces pour identifier les fautes par rapport à Random. Et enfin la stratégie 1-wise possède une meilleure efficacité pour la détection des fautes.

3.2 Évolution de JHipster

En ce jour (Avril 2018), JHipster représente 169 versions depuis sa création [15], une popularité évaluée sur GitHub à 9959 étoiles, une communauté de 400 contributeurs et près de 54000 téléchargements pour le mois de mars 2018 [16]. Cette constante évolution de JHipster est le résultat de l'intégration continue de plusieurs technologies et d'outils permettant de répondre aux besoins des développeurs et à l'évolution technologique. Le tableau 3.1 récapitule les variations des fonctionnalités de JHipster entre la version 3.6.1 et 4.8.2 . On remarque qu'il n'y a que de nouvelles fonctionnalités, les autres restant inchangées. Le nombre de nouvelles fonctionnalités et technologies intégrées à JHipster induit l'augmentation de la variabilité et donc du nombre d'application pouvant être générées mais également de potentiels nouveaux Bugs. Ceci a un effet considérable sur l'effort et le coût de test. En effet l'ajout d'une fonctionnalité à JHipster correspond à la mise à jour du Feature Model de la version courante et donc un effort d'ingénierie conséquent, cela augmente aussi le nombre de configurations générées et par conséquent la quantité de ressource nécessaire pour générer et tester les applications correspondantes.

Features	JHipter 3.6.1	JHipter 4.8.2
MsSql	KO	OK
Angular Translate	KO	OK
Angular4	KO	OK
Eureka	KO	OK
Consul	KO	OK
Session Social	KO	OK
Infinispan	KO	OK
MessageBroker	KO	OK
SwaggerCodegen	KO	OK
CluterHttpSession	KO	OK

Table 3.1 – Feature et évolution de JHipster

3.3 Motivation

Les travaux d'Halin *et al.* [1] ont permis de confronter les différents algorithmes d'échantillonnage à une vérité terrain, que ce soit par rapport à l'ensemble des configurations possibles ou bien par rapport au budget de test de l'équipe, rendant hors de portée la plupart des algorithmes existants. Cependant, comme le montre le tableau 3.1, l'évolution de JHipster montre une augmentation des fonctionnalités. La conséquence est l'accroissement du nombre de configurations et donc une influence directe sur la procédure et le coût de test décrit par Halin *et al.* [1]. Notre motivation se trouve dans l'optique de faire évoluer la procédure de test conjointement à l'évolution de JHipster et d'évaluer ce processus d'évolution. Il est donc question de savoir quel est le coût relatif à la mise à jour des procédures de test décrites par Halin *et al.* [1]. Il est également question de savoir quelles sont les invariants qu'on pourrait observer d'une version à une autre de JHipster, cela pourrait nous permettre d'identifier les stratégies de test les plus efficaces et adaptées à l'évolution de JHipster.

3.4 Méthode et Question de recherche

3.4.1 Questions de recherche

Ces travaux de Master ont pour but de répondre à deux questions de recherche. Nous voulons tout d'abord mesurer le coût en terme d'ingénierie, de ressource temps et matériel pour faire évoluer les procédures de test décrites par Halin *et al.* [1],enfin il est question de déterminer les stratégies de test les plus efficaces et adaptées à l'évolution de JHipster entre deux versions 3.6.1 et 4.8.2, ceci afin de généraliser les résultats de Halin *et al.* Nous avons choisi pour notre cas d'étude, la version 4.8.2 de JHipster car elle était la plus récente au début de nos travaux. Pour atteindre nos objectifs, nous avons découpé nos deux questions de recherche en cinq sous questions auxquelles nous essayerons de répondre dans ce document :

- **RQ 1.1 :** Quel est le coût d'ingénierie pour mettre à jour l'infrastructure de test décrite par Halin *et al.* [1] pour la version 4.8.2 de JHipster ?
- **RQ 1.2 :** Quel est le coût en temps et en ressource nécessaire pour le test de JHipster 4.8.2 ?

Dans l'optique de comparer les résultats de test pour les deux versions et de pouvoir choisir les stratégies de test les plus adaptées, nous essayerons également de répondre aux questions suivantes :

- **RQ 2.1 :** Combien de bugs et quels types de fautes obtenons nous pour JHipster 4.8.2 ?
- **RQ 2.2 :** Les fautes sont elles les mêmes ou sont elles occasionnées par les mêmes interactions de Features pour les deux versions de JHipster ?
- **RQ 2.3 :** Obtenons nous les mêmes résultats avec les techniques d'échantillonnage utilisées par Halin *et al.* [1] ?

3.4.2 Méthodologie

Pour répondre à ces questions, la méthodologie utilisée a consisté à répliquer les travaux de Halin *et al.* [1] sur la version 4.8.2 de JHipster. Ceci nous a permis non seulement de mesurer l'effort et le coût de mise à jour du workflow de test, mais également d'effectuer une étude comparative des résultats obtenus sur les deux versions. Notre méthodologie a donc consisté à suivre les mêmes étapes que celles décrites par Halin *et al.* [1] :

- Extraction de la variabilité
- Mise à jour de l'infrastructure de test et dérivation des configurations
- Génération, Compilation et Exécution des configurations
- Exécution du Workflow
- Analyse quantitative, qualitative et statistique des Bugs
- Étude comparative des résultats

Il est cependant à noter quelques modifications apportées quant à cette méthodologie. En effet pour des raisons que nous expliquerons dans les sections suivantes, nous avons choisi à la phase d'exécution du workflow de créer pour chaque configuration, un environnement contenant tous les outils nécessaires à son exécution pendant du Build. Nous avons également opté pour un nouvel environnement de déploiement et d'exécution des configurations. La figure 3.2 de la section 2.1 reprise de Halin *et al.* [1] résume les différentes étapes de test.

3.4.2.1 Extraction de la variabilité

Cette étape a consisté à extraire la variabilité relative à la version 4.8.2 de JHipster et d'élaborer le Feature Model de JHipster 4.8.2 . Nous avons implémenté le Feature Model par extraction des Features et des contraintes décrites dans les fichiers prompts.js (Server, Client et Application) contenus dans le code du générateur JHipster. En effet, l'exécution de ces fichiers permet à un utilisateur de générer des configurations par simple réponse aux questions qui lui sont posées. Nous avons donc identifié dans ces fichiers, les Features et les contraintes qui nous ont permis de concevoir une première version du Feature Model décrit par la figure 3.6. Nous avons ensuite utilisé Familliar pour implémenter le Feature

Model correspondant et dériver les configurations valides. La figure 3.5 présente quelques exemples de contraintes et Features trouvées dans le fichier server/prompts.js de Jhispter 4.8.2 . Nous observons en effet qu'en fonction des valeurs prises par les variables tels que **databaseType**, **applicationType**, **authenticationType** correspondants respectivement au type de base de données, au type d'application et au type d'authentification choisie par l'utilisateur, le générateur affichera un choix d'option particulier à l'utilisateur. C'est ainsi que le choix d'une base de donnée de type **cassandra** couplé avec une application de type **monolith** ainsi qu'un mode d'authentification **session** ou **jwt** permet à l'utilisateur d'avoir une nouvelle option de choix nommée **Social login** (définissant un mode de connexion à l'application en utilisant un compte sur des reseaux sociaux tels que Google, Facebook, Twitter,..) et dont la valeur est **enableSocialSignIn : true**. Toutes les options ainsi décrites sont des Features à considérer dans la modélisation du Feature Model du Générateur et les contraintes permettent de décrire les combinaisons possibles de Features.

```
if (this.databaseType !== 'cassandra' && applicationType === 'monolith' && (this.authenticationType === 'session' || this.authenticationType === 'jwt')) {
    choices.push(
        {
            name: 'Social login (Google, Facebook, Twitter)',
            value: 'enableSocialSignIn:true'
        }
    );
}
if (this.databaseType === 'sql') {
    choices.push(
        {
            name: 'Search engine using Elasticsearch',
            value: 'searchEngine:elasticsearch'
        }
    );
}
if ((applicationType === 'monolith' || applicationType === 'gateway') &&
        (this.hibernateCache === 'no' || this.hibernateCache === 'hazelcast')) {
    choices.push(
        {
            name: 'Clustered HTTP sessions using Hazelcast',
            value: 'clusteredHttpSession:hazelcast'
        }
    );
}
if (applicationType === 'monolith' || applicationType === 'gateway') {
    choices.push(
        {
            name: 'WebSockets using Spring Websocket',
            value: 'websocket:spring-websocket'
        }
    );
}
choices.push(
    {
        name: 'API first development using swagger-codegen',
        value: 'enableSwaggerCodegen:true'
    }
);
choices.push(
    {
        name: '[BETA] Asynchronous messages using Apache Kafka',
        value: 'messageBroker:kafka'
    }
);
```

FIGURE 3.5 – Exemples de contraintes

Le résultat de cette étape est la construction d'une première version du Feature Model. Cette première version nous donne un nombre total de 887296 configurations pour 58 Features. La figure 3.6 décrit le Feature Model obtenu.

3.4.2.2 Dérivation des configurations associées au Feature Model

Cette partie a consisté à utiliser l'API Familiar de Java pour implémenter le Feature Model et générer les différentes configurations qui lui sont associées. En effet nous avons utilisé le premier module de l'infrastructure de test [4] (JHipsterTest) pour générer les configurations. Pour cela, nous avons commencé par mettre à jour dans l'infrastructure de test le code Familiar décrivant le Feature Model de JHipster. Ceci permettant de passer le Feature Model en entrée à l'infrastructure de test. Nous avons ensuite identifié les nouvelles Features et contraintes relatives à la nouvelle version de JHipster. l'étape suivante a été d'ajouter les nouvelles Features et les traitements associés à ces derniers à la classe modélisant les configurations mais également supprimer les traitements devenus inutiles.
Nous avons enfin ajouté les nouvelles contraintes à la classe qui utilise le Feature Model pour générer les configurations. A la fin de cette phase, comme dans le cas des travaux de Halin *et al.* [1], nous avons réussi à générer les différentes configurations stockées dans des fichiers "yorc-json".

3.4.2.3 Mise à jour de l'infrastructure de test

La mise à jour du workflow de test c'est effectuée en trois étapes :

- **Identification des nouveaux Features** Nous avons identifié pendant cette étape 10 nouvelles fonctionnalités ajoutées entre la version 3.6.1 et 4.8.2 de JHipster. Ce travail étant important dans la mesure où chaque nouvelle fonctionnalité correspond à une nouvelle Feature et donc de manière implicite à l'écriture du code permettant de traiter les configurations qui la possèdent. Nous avons également identifié les Features qui ont été supprimées entre les deux versions de JHipster, ceci nous a

4. https ://github.com/axel-halin/Thesis-JHipster/blob/master/FML-brute

FIGURE 3.6 – Première version du Feature Model.

permis par la suite de supprimer de l'infrastructure de test les traitements devenus inutiles.

- **Mise à jour du feature modele encoder dans le workflow** L'infrastructure de test implémenté par Halin *et al.* [1] prend en entrée le Feature Model conçu pour la version 3.6.1 . Cette étape a consisté à parser le Feature Model obtenu après correction (Figure 3.8) afin de le passer en entrée à l'infrastructure de test.
- **Mise à jour des procedures du workflow** L'évolution de JHipster implique de nouvelles fonctionnalités et donc une influence sur la manière de tester. Les nouvelles fonctionnalités devant être prise en compte, cette étape a consisté à modifier les procédures de test afin qu'elles puissent prendre en compte les nouvelles fonctionnalités ou ignorer les traitements devenus inutiles. Il s'agit des procédures de génération de scripts, de compilation, Build , d'exécution de test et de formatage des résultats.

3.4.2.4 Analyse du Workflow de test et correction des Bugs

Analyse du Workflow : Après avoir mis à jour l'infrastructure de test, nous avons soumis manuellement quelques configurations aux différentes étapes du processus de test (compilation, Build, exécution des tests). En effet, avec la commande "jhipster" exécutée dans le répertoire contenant chaque configuration, nous avons vérifié si la configuration se génère, compile et Build sans erreurs. Si tel n'est pas le cas, nous vérifions la nature des Bugs rencontrés. Cette activité est décrite par la figure 3.7 .

Durant cette phase nous avons identifié trois types de Bugs :

- **Les Bugs causés par des erreurs de modélisation du Feature Model**
- **Les Bugs causés par l'ajout du code dans l'infrastructure de test**
- **Les Bugs causés par le code généré par JHipster**

La détermination des Bugs étant notre sujet central, la suite du travail a été de les identifier pour toutes les configurations dérivées du Feature Model et ensuite de les catégoriser et d'effectuer une analyse de ces derniers. Cependant, il est important de ne traiter que les vrais Bugs d'où la nécessité de corriger ceux causés par le Feature Model ou l'environnement de test.

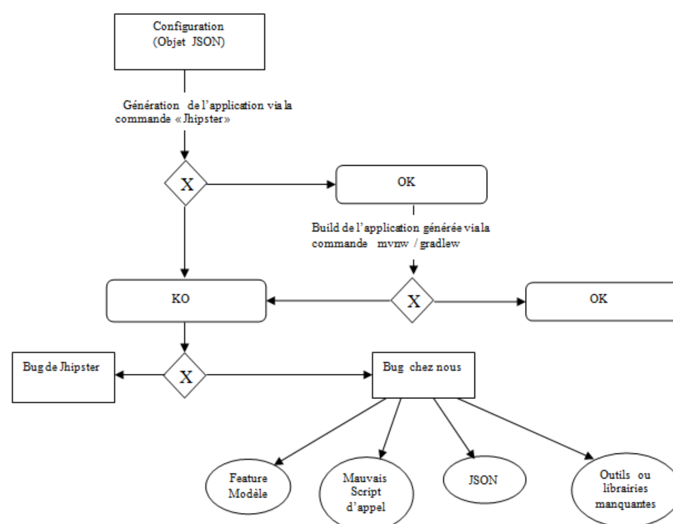

FIGURE 3.7 – Procédure d'exécution manuelle des activités de tests

Correction des Bugs et amélioration du Feature Model : Cette phase a consisté à corriger les Bugs causés par des erreurs de modélisation du Feature Model et donc de mettre à jour ce dernier conjointement aux modifications effectuées. Après correction des erreurs de modélisation, nous avons choisi d'ajouter une nouvelle contrainte au Feature Model. En effet, les trois Features nommées "cucumber", "gatling" et "protractor" étant optionnelles, nous avons choisi de les activer pour toutes les configurations. Cela nous évite alors de tester plusieurs fois la même configuration avec juste des valeurs différentes pour ces Features. Le Feature Model obtenu nous donne un nombre total de 98154 configurations valides et est décrit par la Figure 3.8.

Correction des Bugs liés à L'infrastructure : Il s'agit de corriger le code de l'infrastructure de test pour que cette dernière puisse prendre en compte les changements effectués sur le Feature Model.

Réutilisation du code : Durant l'exécution manuelle du workflow, nous avons remarqué que certaines configurations généraient des erreurs lors de la phase de Build. Ces erreurs n'étaient dues ni à l'infrastructure de test ni au code généré par JHipster, mais plutôt par l'environnement d'exécution (erreur de connexion à un serveur, outils manquant, problème de réseaux etc.). Nous avons également remarqué que JHipster disposait d'image Docker permettant de simuler la plupart d'outils dont a besoin une

configuration pour s'exécuter. Face à ce constat, nous avons décidé de réutiliser le code Docker pour créer un environnement d'exécution spécifique à chaque configuration contenant tous les outils dont elle aura besoin pour son exécution (serveur de base de données, gestionnaire de service, service de gestion de message stream, etc.) . Cela implique une modification du Workflow. La figure 3.9 décrit le nouveau Workflow.

FIGURE 3.8 – Feature Model corrigé.

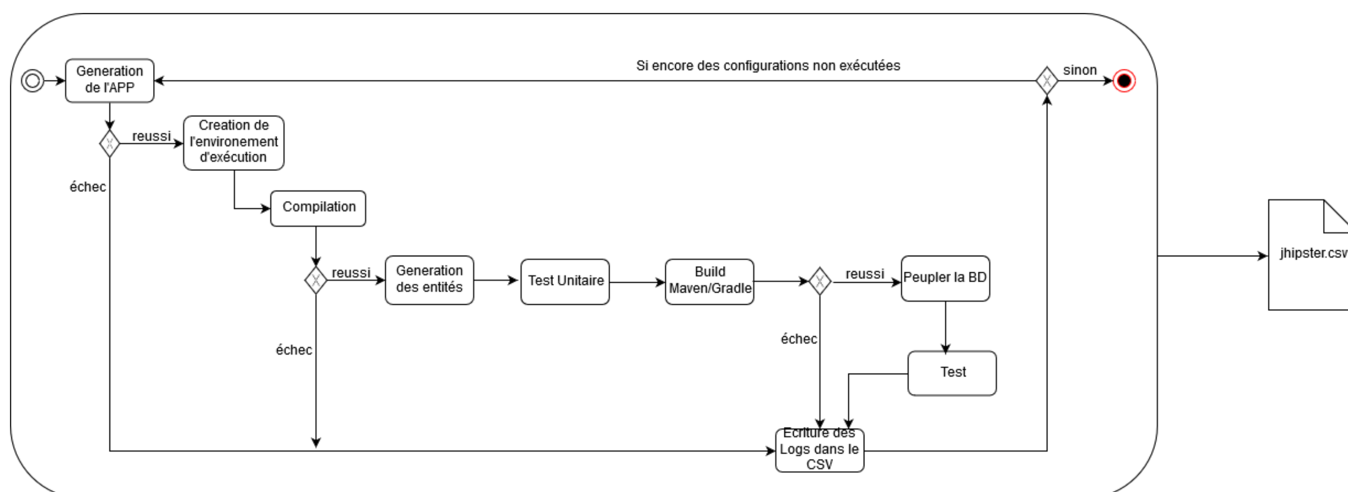

FIGURE 3.9 – Nouveau Workflow de test.

Remarque

Certaines fonctionnalités ne sont pas représentées sur le Feature Model, ceci s'explique par le fait qu'elles ne représentent pas une vraie variabilité du système. En effet, les Features (ServerPort, ModuleToInstall, ModuleNameClient,PackageName, UaaBasename, InternationalizationSupport) qu'on retrouve dans les fichiers Prompt.js de JHipster n'ont pas une grande incidence sur les configurations. Elles peuvent donc tout simplement prendre des valeurs par défaut.

3.4.2.5 Déploiement et exécution du Workflow sur un environnement de Calcul Distribué

Afin de pouvoir tester un grand nombre de configuration, nous avons exécuté toutes les configurations sur la grille de calcul interne à l'IRISA [5] (IGRIDA) [6]. Pour atteindre cet objectif, nous avons écrit des scripts nous permettant de déployer l'infrastructure de test et l'exécuter en boucle sur la grille. Le lien vers les différents scripts est décrit en annexe.

Pourquoi avoir choisi IGRIDA ?

IGRIDA est une grille de calcul mise à la disposition des équipes de recherche de l'IRISA /

5. https ://www.irisa.fr/
6. http ://igrida.gforge.inria.fr/

INRIA, de Rennes. Elle est composée de 138 nœuds de calcul (1700 cœurs) et d'un espace de travail partagé de 3,4 To. Elle offre également 5 nœuds GPU avec GPU NVIDIA [35].

La figure 3.10 présente les caractéristiques de quelques machines d'IGRIDA.

Cluster name	#	CPU	REFERENCE	RAM	DISK	Network
Calda	5	2 x 8 cores Sandy Bridge	Intel(R) Xeon(R) CPU E5-2450 0 @ 2.10GHz	48GB	2 x 600GB	Infiniband + 1GB/s
Lambda	11	2 x 6 cores Westmere-EP	Intel(R) Xeon(R) CPU E5645 @ 2.40GHz	48GB	2 x 600GB	Infiniband + 1GB/s
Flagada	10	2 x 2 cores	Intel(R) Xeon(R) CPU 5140 @ 2.33GHz	4GB	50GB	1 GB/s
Mida	3	2 x 8 cores Sandy Bridge-EP	Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20GHz	64GB	2 x 300GB	1 GB/s
Manda	4	2 x 8 cores Sandy Bridge-EP	Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20GHz	128GB	2 x 300GB	1 GB/s
Panda	10	2 x 4 cores Clovertown	Intel(R) Xeon(R) CPU L5310 0 @ 1.60GHz	8GB	1 x 73GB	1 GB/s
Gouda	8	2 x 4 cores Clovertown	Intel(R) Xeon(R) CPU E5345 0 @ 2.33GHz	8GB	1 x 73GB SAS 15k	1 GB/s
Dalida	20	2 x 4 cores Clovertown	Intel(R) Xeon(R) CPU E5345 0 @ 2.33GHz	8GB	1 x 73GB SAS 10k	1 GB/s
Bermuda	48	2 x 4 cores Gulftown	Intel(R) Xeon(R) CPU E5640 @ 2.67GHz	48GB	2 x 300GB SAS 10k	1 GB/s
Neurinfo1	12	2 x 20 cores (hyperthreading)	Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHzGHz	128GB	2 x 300GB SAS 10k	1 GB/s
Armada	4	2 x 24 cores (hyperthreading)	Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz	192GB	2 x 600GB SAS 10k	1 GB/s

FIGURE 3.10 – Caractéristiques de quelques machines d'IGRIDA [35]

Pour cette étude, nous avons choisi cette infrastructure pour plusieurs raisons :

- **Flexibilité** : Bien que nous avions pour objectif de tester toutes les configurations de JHipster, nous voulions également permettre la distribution des tâches de test. IGRIDA étant une grille interne nous permet d'organiser le test suivant cette méthodologie
- **Récolte de résultat** : IGRIDA permet assez aisément de récupérer les résultats depuis la grille vers une machine locale. Cela évite de passer par le réseau internet pour récupérer les résultats d'exécutions avec pour risques : des coupures de

connexion qui pourraient entraîner la perte des données.

- **Virtualisation et gestion des ressources** : Le déploiement du workflow est assez facilité sur IGRIDA car, elle dispose des images de système déjà prêtes à l'utilisation ce qui facilite l'aspect de la virtualisation des processus. De plus, IGRIDA permet moyennant une limite imposée, de gérer les ressources en fonction des besoins spécifiques ceci nous permet de rendre l'exécution des tests plus configurables (choix du nombre de configuration à exécuter, du nombre de coeur et de la mémoire à utiliser, etc.)
- **Disponibilité** : Bien que possédant moins de machines que le Grid 5000 utilisé par Halin *et al.*, la plateforme IGRIDA est réservée à un petit groupe d'utilisateur à savoir les membres des groupes de recherche de l'IRISA. Elle offre plus de possibilité en terme de disponibilité des machines et permet ainsi d'avoir des machines assez rapidement et de lancer un grand nombre de tâches.

Le processus de génération et d'exécution de toutes les configurations de JHipster sur IGRIDA se découpe en plusieurs étapes :

- **Le déploiement :** consistant à déployer le workflow ainsi que les scripts de génération des configurations sur le compte IGRIDA de l'utilisateur. Cette étape consiste également à déployer les scripts de gestion de l'exécution et la préparation de l'environnement d'exécution des configurations.
- **La dérivation des configurations :** consistant à un appel de fonction qui permettant de dériver toutes les configurations de JHipster. Les différentes configurations sont stockées dans un répertoire sur le compte IGRIDA de l'utilisateur et seront utilisées pour générer et tester les applications.
- **le paramétrage :** consistant à définir les paramètres d'exécution du workflow : le nombre de machine/coeur de calcul, le nombre d'application à tester, la quantité de mémoire allouée pour les calculs, le nombre de temps alloué par machine, ces deux derniers paramètres étant fixés à une valeur par défaut dans ces travaux.
- **L'exécution :** consistant à lancer l'exécution du workflow avec les paramètres définis à l'étape précédente. En effet, une fois le nombre de machine (ici nombre de Job) et le nombre de configurations fournies, le générateur de Jobs génère un nombre de

tâches proportionnelle au nombre de Jobs demandés par l'utilisateur et réparti la charge des configurations de manière équitable à chaque Job. Tous les Jobs exécuteront le même processus à savoir :

1. Copier le nombre de configurations à tester que l'utilisateur a saisi à la phase de configuration. Cette copie se fait à partir du répertoire contenant toutes les configurations dérivées,
2. Exécuter le workflow décrit par la figure 3.9 pour chaque configuration sélectionnée .
3. Stocker les résultats de chaque exécution dans un fichier intermédiare **«fichier.csv»**

- **Ecriture des résultats :** consistant à : écrire le contenue du fichier intermédiaire obtenu dans le fichier résultat **«jhipster.csv»** stocké sur le compte IGRIDA de l'utilisateur et stocker les Logs relatifs à l'exécution de chaque configuration dans un répertoire sur le compte de l'utilisateur.

Les figures 3.11 et 3.12 décrivent le modèle de déploiement et d'exécution du WorkFlow sur IGRIDA. Dans ces figures, l'annotation **WF** représente le workflow décrit à la section 3.4.2.4.

Remarque : Pour exécuter chaque configuration, le workflow crée un environnement d'exécution adapté à la configuration par instantiation d'images Docker correspondant à chaque outil dont a besoin la configuration pour s'exécuter (mysql, kafka, etc.). Ces images sont pré-téléchargées et stockées dans la cache du système d'exploitation virtuel afin d'optimiser le temps de test. Ainsi chaque Job crée une instance du système d'exploitation virtuel et lui fournit les configurations à exécuter. Le système virtuel se chargera alors d'exécuter le workfow sur les configurations fournies et d'écrire les résultats par la suite.

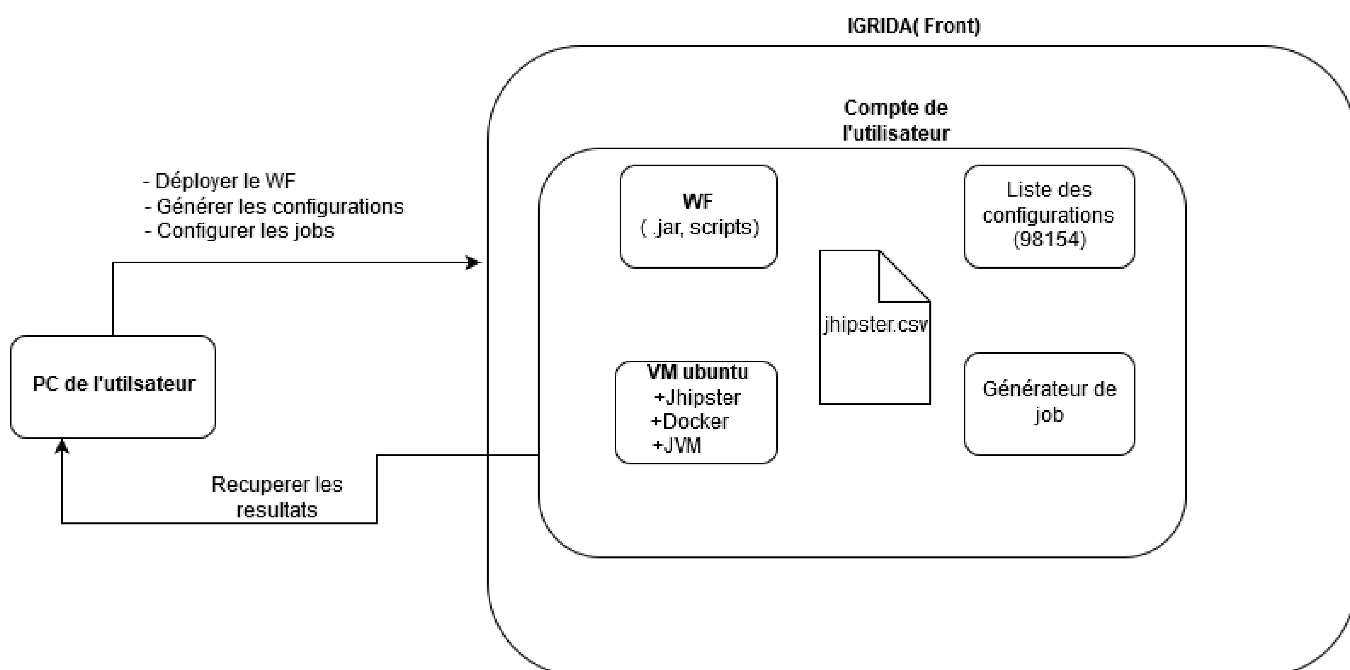

FIGURE 3.11 – Déploiement du Workflow sur IGRIDA

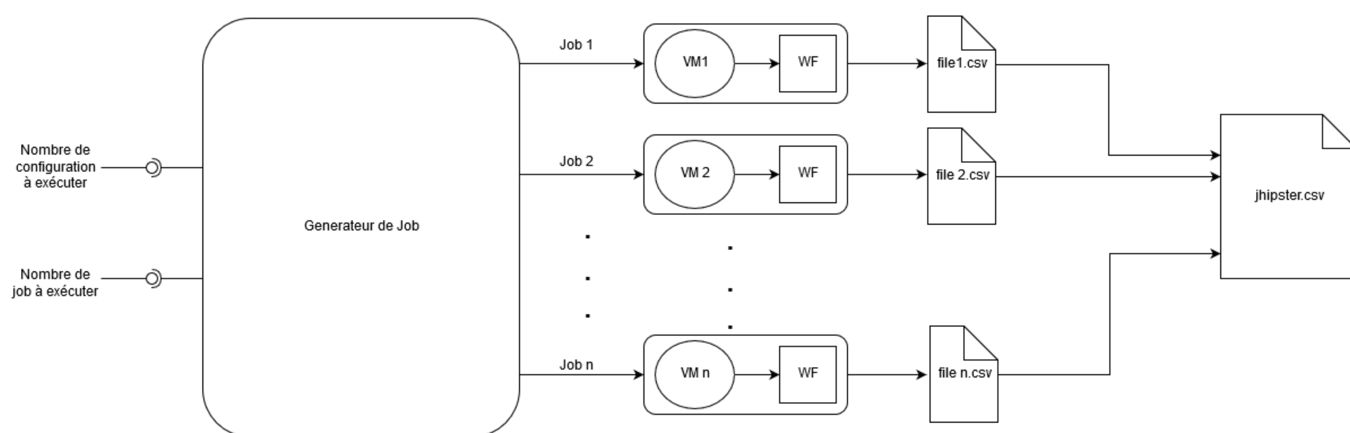

FIGURE 3.12 – Exécution du workflow sur IGRIDA

Chapitre 4

Resultats

Nous présentons dans ce chapitre les résultats obtenus. Ces résultats nous permettent de répondre aux différentes questions de recherche définies à la section3.4.1 du chapitre 2. Nous décrivons d'abord le coût de notre effort d'ingénierie pour la mise à jour et l'automatisation de l'infrastructure de test construite par Halin *et al.* [1] pour toutes les variantes JHipster 4.8.2 (RQ1.1). Nous évaluons ensuite le coût de calcul du test exhaustif (RQ1.2) tout en décrivant les ressources nécessaires (personnes, temps, machines) et en présentant les difficultés rencontrées ainsi que les choix effectués. La deuxième partie de ce chapitre est dédiée à une étude qualitative, quantitative et statistique des Fautes et Bugs obtenus par le test exhaustif. Cette étude permettra de répondre à (RQ2.1) . La dernière partie de ce chapitre sera consacrée à une étude comparative des résultats obtenus pour les deux versions de JHipster (3.6.1 et 4.8.2) ceci afin de répondre à (RQ2.2) et (RQ2.3)

4.1 Le coût du test

4.1.1 Effort d'ingénierie

Le test de toutes les variantes de JHipster 4.8.2 a requis un certain effort d'ingénierie. Cet effort se rapporte essentiellement aux étapes décrites au chapitre 2. Il s'agit à titre de rappel de : L'extraction de la variabilité, la mise à jour du code de l'infrastructure et l'exécution du workflow.
Afin d'extraire la variabilité, il nous a fallu tout d'abord comprendre le fonctionnement JHipster ceci par analyse des fichiers Promt.js et server.js fournis dans le code du générateur. Après analyse, au moyen des contraintes et des Features identifiées, nous avons élaboré une première version du Feature Model. Cette étape nous a pris **0.75 Homme - mois**. Afin de mettre à jour l'infrastructure de test, nous avons commencé par encoder le nouveau Feature Model afin de le passer en entrée à l'infrastructure. Nous avons ensuite ajouté de nouveaux traitements et des modifications du code afin de s'assurer que l'infrastructure effectue les traitements adéquats et adaptés au Feature Model (dérivation des configurations, génération des scripts permettant la construction, la compilation et le Build) des applications correspondantes aux configurations dérivées. Cette étape nous a pris **1 Homme - Mois**. La mise à jour du workflow nous a permis après quelques exécutions manuelles de découvrir un certain nombre de Bugs. Les causes de ces Bugs pouvant être liées au Feature Model, à l'infrastructure ou au générateur JHipster. Cette étape nous à permis d'identifier ces Bugs et de les corriger dans le cas où cela est possible. Ceci nous a abouti à la mise à jour du Feature Model et l'obtention d'une deuxième version qui sera celle utilisée dans la suite. Nous avons également durant cette phase généré et déployé toutes les configurations sur IGRIDA, ceci inclut l'apprentissage de l'environnement IGRIDA et l'écriture des scripts de déploiement. Cette étape nous a coûté **1 Homme - Mois**. Nous pouvons conclure en affirmant que, l'effort d'ingénierie requis pour tester JHipster tout en considérant les travaux déjà effectués par Halin *et al.* [1] est dépendant de plusieurs facteurs : l'analyse et la compréhension de la version de JHipster considérée, le temps de compréhension du fonctionnement de l'infrastructure et enfin du nombre de traitements

à mettre à jour dans le code de l'infrastructure. Ce dernier facteur étant dépendant du nombre de Features et de contraintes obtenues après extraction de la variabilité. Cependant, Halin *et al.* [1] ont déjà implémenté durant leurs travaux plusieurs traitements se rapportant à des Features communes aux deux versions. Étant donné le fait qu'entre la version 3.6.1 et 4.8.2 de JHipster, la plupart des Features et des contraintes restent invariantes, l'effort d'ingénierie se voit donc diminué du fait de la réutilisation du code. Il est toutefois à noter que le déploiement et l'exécution de l'infrastructure sur IGRIDA a coûté énormément de temps, ceci s'explique par le fait que nous avons développé des scripts de déploiement et de récolte des résultats d'exécutions.

RQ 1.1 : Quel est le coût d'ingénierie pour mettre à jour l'infrastructure de test ?

L'effort d'ingénierie pour la mise à jour et le test de toutes les variantes de JHipster 4.8.2 est de **2.75 Homme-Mois**

4.1.2 Coût en ressource de calcul

Pour des raisons décrites au chapitre 3, nous avons choisi de déployer et d'exécuter le workflow de test sur IGRIDA. L'exécution se découpe en plusieurs étapes : la génération des applications, la compilation et l'exécution des applications générées. Il est à rappeler que le Feature Model obtenu nous a permis de dériver 98154 configurations valides, ce nombre énorme de configuration requiert beaucoup de temps pour la génération, la compilation, le Build et l'exécution de test. Compte tenu de cette remarque et afin de réduire le temps d'attente des résultats, nous avons choisi de ne pas exécuter les tests unitaires. On peut aussi remarquer le fait que l'exécution des configurations sur IGRIDA se fait en utilisant une instance du système d'exploitation virtuel et un nombre limité de configuration par chaque Job, l'espace de stockage utile pour le test se limite à celui contenant toutes les

configurations , les scripts et le fichier résultat et le système d'exploitation virtuel. L'espace de stockage alloué pour le compte utilisateur soit 3,5 To est de manière générale assez suffisant, ceci se justifie par le modèle d'exécution que nous avons choisi. En effet, les configurations sont exécutées sur des instances virtuelles de systèmes d'exploitations, elles utilisent donc la mémoire allouer pour stocker les fichiers créer, cependant après écriture dans le fichier resultat et sauvegarde des Logs, les fichiers créés sont supprimés. L'espace requis est donc l'espace nécessaire pour stocker les 98154 configurations (fichier json) + la mémoire utile pour stocker les Logs et le fichier resultat.

Chaque machine IGRIDA possède en moyenne 6 coeurs de calcul et le temps d'attente pour la réservation d'une machine complète est très élevé. Cependant, ce temps est assez réduit pour la reservation des coeurs de calculs moyenant un nombre inférieur à 6. Pour pouvoir tester les 98154 configurations tout en tenant compte de la disponibilité des machines, nous avons choisi d'utiliser des Jobs pour les exécuter sur des coeurs de calcul à raison de 100 configurations par Job et d'associer 2 coeurs à chaque Job. Avec ces paramètres et hormis le temps d'attente, nous avons pu exécuter complètement 1 job (100 configurations) en 12 heures. Cependant, nous avons effectué 20 périodes de 12 heures et de 50 jobs [1] chacune, ce qui équivaut à 240 heures pour 17 machines. Au totale, nous avons eu besoin de 4080 heure-machine pour exécuter les 98154 configurations

RQ 1.2 : Quel est le coût en temps et en ressource nécessaire pour le test de JHipster 4.8.2 ?

Suivant l'infrastructure et les paramètres décrits(2 coeurs de calcul pour 100 configurations), le test de JHipster 4.8.2 nécessite en moyenne **4080 heure-machine** et **3.5 To** d'espace de stockage.

1. (50 jobs = 100 coeurs, 1 machine = 6 coeurs) -> 50 jobs = 17 machines

4.2 Analyse des Bugs

L'exécution du workflow sur les configurations obtenues produit comme résultat un fichier CSV. Ce fichier contient pour chaque configuration les résultats de la génération, la compilation et le Build de l'application correspondante. On peut donc observer dans ces fichiers les différents Bugs relatifs à chaque configuration. L'analyse du fichier obtenu nous permettra de répondre aux questions de recherche. En effet une analyse statistique (quantitative et qualitative) du fichier CSV nous permettra de répondre à **RQ2.1** ensuite une analyse des résultats des techniques d'échantillonnage sur le fichier, suivie d'une étude comparative avec les résultats de Halin *et al.* [1] nous permettra de répondre à **RQ2.2** et **RQ2.3**.

4.2.1 Analyse statistique quantitative

Nous avons obtenu pour cette version de JHipster le nombre de 98154 configurations. Ces configurations ont été testées au moyen du Worflow décrit au chapitre précédent. Sur les **98154** configurations testées, nous avons obtenu un pourcentage de **20.59%** soit **20209** configurations qui rencontrent des problèmes à l'exécution. Ce taux d'échec est observé durant deux phases d'exécutions du processus : à la compilation avec **249** configurations et à la phase de Build avec **19960** configurations. Il est aussi à noter que toutes les configurations ont été générées sans échec.

Tout comme Halin *et al.* [1] nous avons observé les données résultats sur la base des types d'applications et des modes d'authentifications et le résultat est présenté par la figure 4.1. En effet, pour les types d'application, nous avons observé une similitude entre les applications de type Gateway et microservice, avec respectivement **54,5%** des applications de type Gateway soit **10497 configurations** et **54,6%** des applications microservice, soit **2533 configurations**. Cependant, il y a moins de similitudes pour les autres types d'applications : **10,6% (6766 configurations)** pour les applications monolithique et **17,9% (413 configurations)** pour serveurs UAA. L'étude des modes d'authentification quant à elle montre que les applications dont le mode d'authentification est JWT, Oauth2 et Session ont approximativement le même pourcentage de Bugs , il s'agit respectivement de :

10,4% (5053 configurations), **10,3% (943 configurations)** et **10,7% (1937 configurations)** par contre les applications ayant comme mode d'authentification UAA sont celles avec le plus d'échecs. En effet pour ces applications, on observe un taux d'échec égal à **86,7%** soit un total de **12276 configurations)**.

Cette analyse nous donne des informations sur le nombre de Bugs en regard des différentes Features, mais elle ne nous en donne aucune sur la nature des Bugs ou les causes potentielles. Pour compléter la réponse nous effectuons dans la section suivante une analyse qualitative.

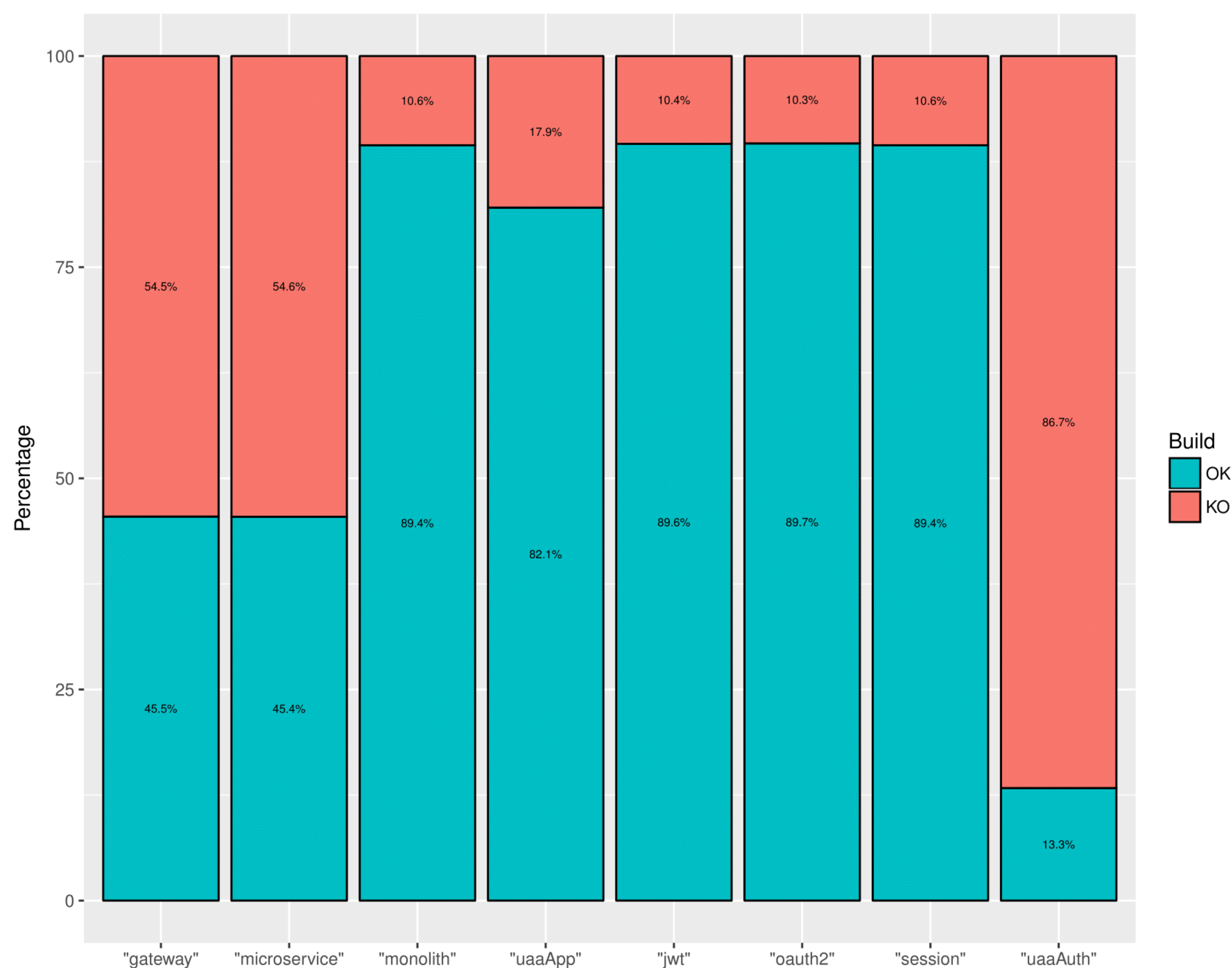

FIGURE 4.1 – Proportions d'échec de Build par feature

4.2.2 Analyse statistique qualitative

Le but de l'analyse qualitative a consisté à extraire les types de fautes rencontrées dans les données ainsi que d'établir une relation entre elles et les différentes Features. Cette méthode nous a permis de catégoriser les Bugs et de déterminer les différentes interactions de Feautures les produisant. Pour y arriver, nous avons utilisé la même technique que Halin *et al.* [1]. En effet, nous avons utilisé la méthode d'apprentissage par la règle d'association de Hahsler *et al.* [36]. Cette méthode vise à extraire les relations entre des variables d'un data set. Une règle d'association est constituée d'un antécédent (LHS) et d'un conséquent (RHS). l'antécédent représente l'observation faite sur les données entraînant de manière générale le conséquent. Ainsi le but d'utiliser la technique d'apprentissage est de déterminer à partir du fichier CSV, des règles permettant d'associer des combinaisons de Features (LHS) à un résultat de compilation ou de Build ayant échoué (RHS). Pour cette expérience, nous avons utilisé comme support le package *arules* [2] de **R** et considéré les mêmes paramètres que Halin *et al.* [1]. En effet, les paramètres fixés permettent d'obtenir des règles pour lesquelles le conséquent (RHS) aura soit la valeur Build= "KO" ou Complie="KO". Nous avons également choisi de rechercher des antécédents contenant 1,2,3 ou 4 Features. Le tableau 4.1 présente les différentes règles obtenues ainsi que les proportions de données couvertes par l'antécédent et par le conséquent. Sur base de ces règles et par généralisation, nous avons pu extraire 14 règles décrivant chacune les interactions de deux ou trois Features.

A partir des règles obtenues, nous avons pu identifier et catégoriser les Bugs. En effet nous avons généralisé certaines règles d'associations obtenues afin d'observer 11 catégories d'erreurs. Les trois règles de généralisations appliquées sont décrites comme suite :

1. **Regle 1** : DevDatabaseType="mssql",ServiceDiscoveryType="eureka" -> Build="KO"
   Regle 2 : DevDatabaseType="mssql", ServiceDiscoveryType="consul" -> Build="KO"
   Bug Resultant : **MssqlWithServiceDiscovery**
2. **Regle 1** :DevDatabaseType="mssql", HibernateCache="ehcache" -> Build="KO"
   Regle 2 : DevDatabaseType="mssql", HibernateCache="infinispan"-> Build="KO"
   Bug Resultant : **MssqlWithHibernateCache**

2. https ://cran.r-project.org/web/packages/arules/index.html

LHS	RHS	Proportion
AuthenticationType=uaa, ApplicationType=monolith	Build=KO	123
AuthenticationType=uaa, ClusteredHttpSession=hazelcast	Build=KO	82
AuthenticationType=uaa, DatabaseType=no	Build=KO	8
AuthenticationType=uaa,Websocket=spring-websocket	Build=KO	8424
AuthenticationType=session, DevDatabaseType=mssql	Build=KO	459
AuthenticationType=oauth2, DevDatabaseType=mssql	Build=KO	222
ApplicationType=gateway, ProdDatabaseType=mongodb BuildTool=gradle	Build=KO	559
DevDatabaseType=mssql, EnableSocialSignIn	Build=KO	217
DevDatabaseType=mssql,ServiceDiscoveryType=eureka consul	Build=KO	5857
DevDatabaseType=mssql, HibernateCache=ehcache	Build=KO	984
DevDatabaseType=mssql, HibernateCache=infinispan	Build=KO	967
HibernateCache=hazelcast, ProdDatabaseType=mongodb BuildTool=gradle	Build=KO	360

TABLE 4.1 – Règles d'association généralisées pour les échecs de compilation et de Build

3. **Regle 1** :HibernateCache="hazelcast", ProdDatabaseType="mongodb" BuildTool="gradle" -> Build="KO"

   Regle 2 : ApplicationType="gateway", ProdDatabaseType="mongodb" BuildTool="gradle"-> Build="KO"

   Bug Resultant : **MongodbWithGradle**

Les différentes classes d'erreurs obtenues couvrent **96.7%** des configurations à Bugs observées et la figure 4.2 présente les proportions observées pour chacune d'elles.

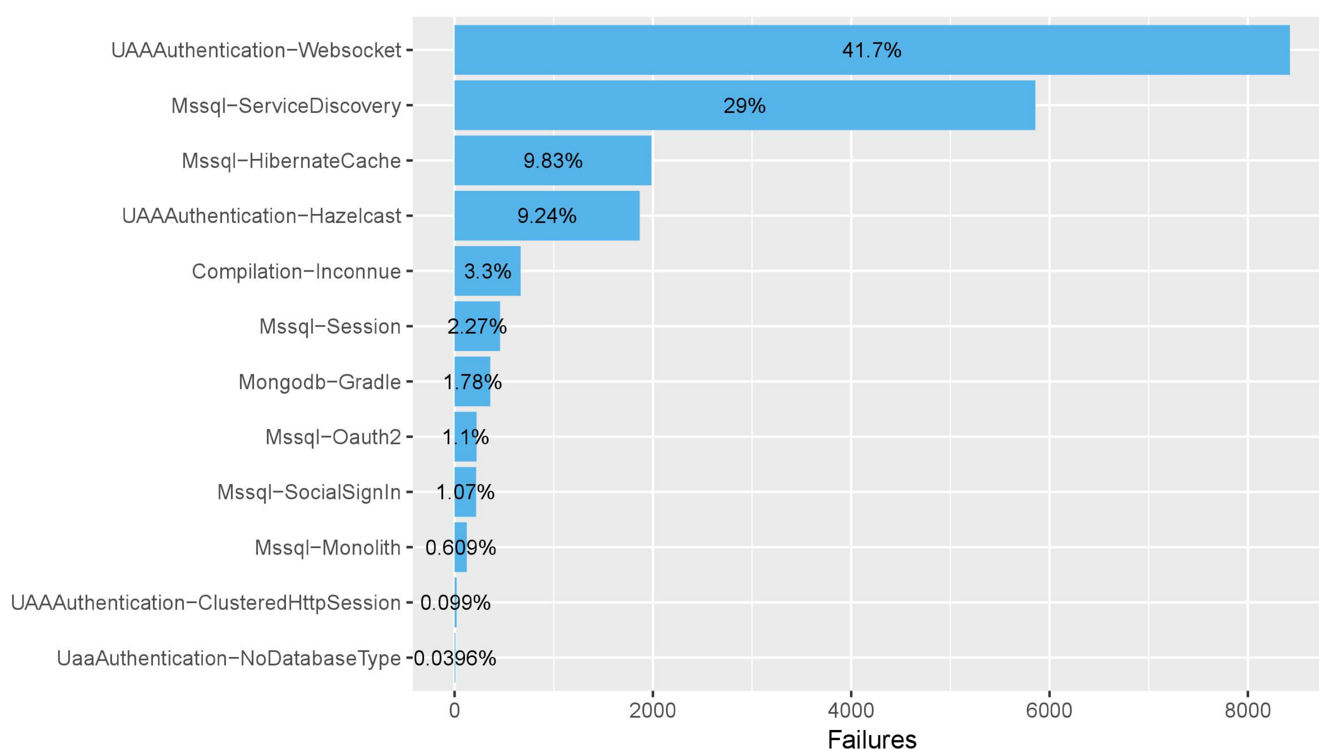

FIGURE 4.2 – Proportions d'échec par classe de fautes

Remarque : le pourcentage de 3,3% obtenu pour des erreurs inconnues est du à trois causes :

— **Les échecs de compilations** : très peu de configurations sont en échecs de compilations (249 configurations soit 1.23% Bugs), l'algorithme de détermination des règles d'associations n'a pas réussi à construire des règles sur base de ces configurations. Il ne les a donc pas pris en compte.

— **Les erreurs de généralisation des règles d'associations** : nous avons généralisé certaines règles d'associations, afin de regrouper les fautes au sein des classes plus générales et donc de réduire le nombre de règles et de classes. Cette stratégie pourrait probablement générer des erreurs.

— **Les erreurs liées à l'environnement de test** : En effet l'environnement de test est susceptible de générer des erreurs. Nous pensons que ces fautes font parties du pourcentage inconnu de Bugs non couvert par les règles d'associations

Nous avons ainsi obtenu les catégories de Bugs suivantes :

#UAAAuthentication-Hazelcast : Classe des Bugs observés par association de la méthode d'authentification "uaa" à l'activation de la valeur "hazelcast" du Feature HibernateCache.

#UAAAuthentication-ClusteredHttpSession : Classe des Bugs observés par association de la méthode d'authentification "uaa" à l'activation de la Feature ClusteredHttpSession.

#UaaAuthentication-NoDatabaseType : Classe des Bugs observés par association de la méthode d'authentification "uaa" sans base de données.

#Mssql-Monolith : Classe des Bugs observés par association d'une application de type monolithic avec une base de données de type Microsoft Sql.

#Mssql-SocialSignIn : Classe des Bugs observés par association d'une base de données de type Microsoft Sql et l'activation de la Feature SocialSignIn (ces deux Features sont des nouveautés de la version de JHispter étudiée).

#UAAAuthentication-Websocket : Classe des Bugs observées par association de la méthode d'authentification "uaa" et l'activation du Feature Websocket.

#Mssql-ServiceDiscovery : Classe des Bugs observées par association d'une base de données de type Microsoft Sql et l'une ou l'autre valeur de la Feature ServiceDiscovery (ces deux Features sont des nouveautés de la version de JHispter étudiée).

#Mssql-Session : Classe des Bugs observés par association de la methode d'authentification "session" à une base de données de type Microsoft Sql.

#Mssql-Oauth2 : Classe des Bugs observés par association de la méthode d'authentification "Oauth2" à une base de données de type Microsoft Sql.

#Mssql-HibernateCache : Classe des Bugs observés par association d'une base de données de type Microsoft Sql à l'activation d'une des valeurs *infinispan,ehcash* de la Feature HibernateCache.

#Mongodb-Gradle : Classe des Bugs observées par association d'une base de données de type MongoDB à la valeur "gradle" de la Feature BuildTools.

#Compilation-Inconnue : Classe des Bugs issus soit de la compilation (aucune règle obtenue pour les erreurs de compilations), soit de l'environnement de test ou l'erreur

de généralisation des règles d'associations.

RQ 2.1 : Combien de bugs et quels types de fautes obtenons nous pour JHipster 4.8.2 ?

Le test exhaustif de JHipster 4.8.2 montre un taux d'échec de **20.59%**. Notre analyse à déterminé 11 classes de fautes d'interactions (occasionnées par des interactions de 2 à 3 features) couvrant **96.7%** des erreurs rencontrées pour cette version de JHipster. Les **3.3%** d'erreurs restant sont attribuées aux erreurs de compilation , aux erreurs liées à l'environnement de test et aux erreurs de généralisation des règles d'associations.

4.2.3 Analyse de résultat de sampling

Le test exhaustif nous a permis de déterminer les configurations à échecs mais également de pouvoir les catégoriser. Nous avons ainsi pu déterminer 11 catégories de fautes différentes. Nous avons ensuite appliqué les stratégies d'échantillonnage décrites précédemment (Random, T-wise avec T ∈ {1,2,3,4}, Dissimilarity) afin de déterminer celles qui construisent des échantillons permettant de détecter de manière optimale les erreurs et les différentes classes de fautes identifiées. Nous avons exécuté les différents algorithmes en définissant pour chacun la même taille d'échantillon attendu. Le tableau 4.2 présente les résultats obtenus.

Echantillonage par interaction combinatoire Pour mettre en place la stratégie ICT décrite au chapitre 1 section 2.4.2.1, nous avons généré les échantillons respectant le critère T-wise avec T ∈{1,2,3,4} en utilisant l'outil SPLCAT[23]. L'exécution de SPLACAT pour les 4 variantes nous a permis d'obtenir 4 échantillons correspondants respectivement aux critères 1-wise, 2-wise, 3-wise et 4-wise d'une taille respective de *8,44,197 et 638* configurations. L'échantillon construit avec 1-wise ne permet l'identification d'aucune classe de fautes tandis que celui construit avec 2-wise permet d'identifier 4 classes de fautes sur 11, 3-wise n'en identifie que 5 sur 11 et 4-wise 6 sur 11. En ce qui concerne les proportions d'échecs, on remarque que 1-wise détecte 1 faute sur les 8 configurations constituant l'échantillon tandis que 4-wise détecte 208 pour un échantillon de 638 configurations.

Stratégie	Taille échantillon	Nombre Failure	σ des Failure	Efficacité Failure	Fautes	σ des Fautes	Efficacité Fautes
Random(8)	8	1.71	1.15	21.38%	0.51	0.61	6.38%
PLEDGE(8)	8	2.37	1.04	29.63%	0.71	0.57	8.88%
1-wise	8	1	NA	12.5%	0	NA	0
Random(12)	12	2.67	1.42	22.25%	0.85	0.78	7.08%
PLEDGE(12)	12	3.47	1.42	28.91%	0.98	0.62	8.17%
Random(44)	44	10.16	2.85	23.09%	2.19	0.88	4.98%
PLEDGE(44)	44	13.26	2.56	30.14%	2.54	0.82	5.77%
2-wise	44	11	NA	25%	4	NA	9.09%
Random(197)	197	44.79	5.34	22.74%	4.33	1.08	2.2%
PLEDGE(197)	197	60.29	5.43	30.6%	4.08	0.84	2.07%
3-wise	197	65	NA	32.99%	5	NA	2.54%
Random(638)	638	141.76	9.18	22.22%	6.3	0.98	0.99%
PLEDGE(638)	638	198.06	9.68	31.04%	5.51	0.82	0.86%
4-wise	638	208	NA	32.60188	6	NA	0.94
ALL	98154	20209	NA	20.59	11	NA	0.011%

Table 4.2 – Résultat des strategies d'échantillonnage

Echantillonage basé sur la Dissimilarité Pour créer des échantillons sur base de la stratégie de dissimilarité comme décrite au chapitre 1 section 2.4.2.2, nous avons utilisé l'outil PLEDGE[30]. Afin de pouvoir comparer les algorithmes sur la même taille d'échantillons, Nous avons fixé les tailles des échantillons en fonction des valeurs obtenues pour les T-wise c'est-à-dire 4 échantillons possédant les tailles respectives 8,44,197 et 638, nous avons également créé des séries de 100 échantillons pour chaque cas. Nous avons obtenu avec cette technique les moyennes respectives de 1,3,4 et 6 fautes identifiées par taille d'échantillon.

Echantillonage aléatoire(Random sampling) Pour cette stratégie, nous avons créé 100 échantillons aléatoires différents de 8,44,197 et 638 configurations. L'analyse de

chaque groupe d'échantillons nous a permis d'observer une moyenne de 1, 2, 4 et 6 respectivement pour 8, 44, 197 et 638 configurations.

Les résultats obtenus et présentés dans le tableau 4.2, nous permet de faire plusieurs observations. La première observation est que pour chaque stratégie, le nombre de fautes détectées croît avec la taille de l'échantillon. Cependant, la technique T-wise est la plus efficace pour détecter les fautes. En effet elle possède le meilleur ratio nombre de fautes détectées par rapport à la taille de l'échantillon. La stratégie PLEDGE quant à elle représente la meilleure stratégie pour la détection des échecs, elle est également plus appropriée que Random pour la détection des fautes pour des échantillons de petites tailles. On remarque que pour des échantillons de grandes tailles, la stratégie Random est supérieure à PLEDGE relativement aux nombres de fautes détectées, mais reste inférieure en ce qui concerne la détection des échecs.

4.3 Analyse comparative

Afin de répondre aux questions RQ 2.1 et RQ 2.2 nous analysons dans cette section les résultats obtenus pour la version 4.8.2 de JHipster comparativement à ceux obtenus par Halin *et al.* [1] pour la version 3.6.1. Les tableaux 4.3 et 4.4 permettent d'établir une comparaison entre les deux cas d'études.

4.3.1 L'éffort d'ingénierie

L'effort d'ingénierie déployé pour la mise à jour et le test exhaustif des variantes de JHipster 4.8.2 est de **2.75 Homme-Mois contre 8 Homme-Mois** obtenu par Halin *et al.* pour la version 3.6.1. Ce résultat s'explique par le fait que nous avons réutilisé le code écrit par Halin *et al.*. En réalité, l'effort décrit est relatif à la modélisation du Feature Model, la mise à jour du code de l'infrastructure et l'écriture des scripts de déploiement du workflow sur la grille. En conclusion, étant donné l'existence de l'infrastructure décrite par Halin *et al.*, le coût d'ingénierie relatif au test d'une autre version de JHipster peut être réduit de moitié

et est alors dépendant de l'effort de modélisation de la variabilité, de mise à jour du code de l'infrastructure et du déploiement sur la grille. Nous remarquons aussi une diminution du coût des ressources malgré l'augmentation du nombre de configurations. En effet l'exécution des 98154 configurations requiert 4080 heure-machine et 3.5 To d'espace de stockage contre 4376 heure-machine et 5.2 To d'espace de stockage pour la version 3.6.1 de JHipster. Ce résultat s'explique pour le temps de calcul par le fait que pour chaque configuration, nous avons désactivé les tests et exécuté uniquement la génération, la compilation et le Build,nous avons également utilisé Docker pour toutes les configurations ce qui a réduit considérablement le temps d'exécution. Pour l'espace de stockage, le model d'exécution que nous avons défini nous permet d'exécuter les configurations sans sauvegarder les fichiers créés mais juste les resultats et les Logs pour chaque exécution d'ou le coût de stockage reduit.

On peut donc conclure que IGRIDA est plus intéressant(flexible et disponible) que GRID5000 pour tester de manière exhaustive si on effectue des tâches courtes et en occurrence si les tests sont désactivés dans le workflow.

Cas d'étude	Configurations valides	Coût ingénierie	Coût ressource et temps
JHipster 3.6.1	26256	8 homme-mois	4376 Heure-machine pour 5.2 To d'espace de stockage
JHipster 4.8.2	98154	2.75 homme-mois	4080 Heure-machine et 3.5 To d'espace de stockage

Table 4.3 – Comparaison : coût du test

4.3.2 Taux et types de bugs

L'analyse des résultats du test montre un taux de 20.59% de configurations portant des échecs soit à la phase de compilation ou de Build contre 34.37% pour la version 3.6.1 de JHipster. Nous faisons également l'observation que le nombre de classe de fautes a augmenté et est passé de 6 à 11. Tout comme pour la version 3.6.1, les fautes détectées sont occasionnées par des interactions de 2 ou 3 Features. Aucun invariant n'est trouvé entre les différentes classes de Bugs. Par contre, on observe une similitude dans la nature des Bugs. En effet, un grand nombre de fautes est occasionné par l'interaction de la Feature "authenticationType =uaa" avec d'autres Features, mais aussi l'interaction des Features relatives au type de base données.

Cas d'étude	Taux de Bug	Classes Fautes	Bugs invariants
JHipster 3.6.1	34.37%	6 fautes d'interactions	NA
JHipster 4.8.2	20.59%	11 fautes d'interactions	NA

Table 4.4 – Comparaison : quantité et qualité des Bugs

RQ 2.2 : Les fautes sont elles les mêmes ou sont elles occasionnées par les mêmes interactions de Feature pour les deux versions de JHipster ? Nous avons determiné pour JHipster 4.8.2, 11 classes de fautes couvrant 96.7% des Bugs déterminés par le test exhaustif. Comparativement aux 6 classes de fautes déterminées pour la version 3.6.1, il n'existe pas de classes commune(classe des fautes occasionnées par les interactions des mêmes Features). On remarque toutefois que : pour les deux versions, l'interaction de la Feature *authenticationType=uaa* avec d'autres Features produit la plupart des erreurs (50.7% pour la version 4.8.2 et 35.3% pour la version 3.6.1). On remarque également que l'interaction entre deux nouvelles Features que sont *mssql* et *serviceDiscoveryType* est cause de Bugs (29% des Bugs).

4.3.3 Efficacité des méthodes de sampling

Après analyse des résultats de l'échantillonnage, nous avons pu conclure que pour des échantillons de petites tailles, PLEDGE était plus intéressant que Random pour la détection des fautes cela confirme les résultats obtenus par Halin *et al.* [1]. Nous avons également conclu que Random est la méthode la plus adaptée tant pour la détection des fautes que des échecs. Ceci se justifie par le fait que pour les différentes tailles d'échantillons, Random a une moyenne au-dessus des autres stratégies. Ces resultats confirme ceux obtenu par Halin *et al.* ; on se rend cependant compte que : contrairement à Halin *et al.* 2-wise et 3-wise sont moins efficaces pour déterminer les fautes que Random.

RQ 2.3 : Obtenons nous les mêmes résultats avec les techniques de sampling utilisées par Halin *et al.* [1] ?

Les résultats obtenus pour les deux versions de JHipster confirment deux choses :

- Random est la plus efficace pour la détection des échecs et des fautes.
- Pour des échantillons de petite taille, la stratégie PLEDGE est mieux adaptée pour la détection des échecs et des fautes.
- Pour des échantillons de grandes tailles, 4-wise est la meilleure stratégie pour la detection des configurations en échecs.

Ces résultats permettent également d'émettre des réserves relatives aux résultats de la technique T-wise car :

- Contrairement à la version 3.6.1 de JHipster, 2-wise et 3-wise sont moins efficaces que Random pour déterminer les fautes

On peut conclure que malgré l'évolution de JHispter entre les versions 3.6.1 et 4.8.2, si le budget de test des développeurs restait inchangé à savoir 12 configurations [1], alors la méthode PLEDGE restera la plus adaptée pour le choix des configurations à tester. En effet pour une taille d'échantillon égale à 12, cette stratégie permet de détecter plus de fautes et d'échecs que les autres. Ce résultat vient conforter ceux obtenus par Halin *et al.*

Validité

Les menaces à la validité de nos travaux sont liées à plusieurs facteurs. En effet, l'une des menaces est le fait que, par manque de temps, nous n'avons pas pu implémenter les algorithmes *One-Enabled*, *One-Disabled* et *Most-enabled-disabled*. Ceci pourrait constituer un manquement et biaiser l'étude comparative effectuée. Dans la même lancée, l'exécution (Compilation et Build) des configurations c'est fait sans l'activation des tests unitaires et uniquement avec l'utilisation de Docker ce qui peut biaiser le coût en terme d'utilisation des ressources. Une seconde menace de la validité est le fait que, par manque de temps, nous n'avons pas pu dresser la liste des Bugs de JHipster 4.8.2 répertoriée sur GitHub afin de la comparer aux classes de Bugs obtenues. Cela aurait permis de valider les résultats du test exhaustif. Notons également que nous avons obtenu 3,3% de Bugs non catégorisés dont 1,3% étaient liés à la compilation et le reste (2%) attribués aux erreurs de généralisations de la classification et à l'environnement. Cependant, ce taux élevé d'incertitude représente un facteur à prendre en compte dans la validité des travaux.

Chapitre 5

Conclusion

Nous concluons en résumant les différents résultats obtenus. En effet, la première partie de ces travaux a consisté à décrire le contexte de notre étude. Nous avons décrit la notion de ligne de produits logiciel et donc celle de système hautement configurable tout en montrant l'impact de leur évolution sur leur processus de développement et de test. Nous avons également présenté quelques méthodes de test des lignes de produits logiciels avant d'introduire et présenter les résultats obtenus par Halin *et al.* pour le test d'un cas particulier de système configurable (JHipster 3.6.1). Le but de nos travaux étant de décrire l'impact de l'évolution d'un système configurable (dans notre cas JHipster) sur la stratégie de test à adopter, nous avons dupliqué les travaux de Halin *et al.* sur la version 4.8.2 de JHipster afin de comparer les résultats obtenus et pouvoir décrire la stratégie de test la mieux adaptée en dépit de l'évolution observée du système.

Durant ce travail de duplication, nous avons rencontré plusieurs difficultés. Ces difficultés sont liées à deux grands facteurs : la compréhension de l'environnement (technologies utilisées) et du fonctionnement de JHipster ceci dans le but d'élaborer un Feature Model décrivant au mieux le système et la compréhension puis la maitrise de l'environnement de déploiement et d'exécution du workflow de test ceci de créer un modèle d'exécution adaptés compte tenu du nombre important de configurations dérivées.

Malgré les difficultés rencontrées, nous avons réussi à obtenir plusieurs résultats. Ces résultats se sont attelés à répondre à 5 questions de recherche :

- **RQ 1.1 :** Quel est le coût d'ingénierie pour mettre à jour l'infrastructure de test décrit par Halin *et al.* [1] pour la version 4.8.2 de JHipster ?
- **RQ 1.2 :** Quel est le coût en temps et en ressource nécessaire pour le test de JHipster 4.8.2 ?
- **RQ 2.1 :** Combien de Bugs et quels types de fautes obtenons nous pour JHipster 4.8.2 ?
- **RQ 2.2 :** Les fautes sont elles les mêmes ? Sont elles occasionnées par les interactions des mêmes Features pour les deux versions de JHipster ?
- **RQ 2.3 :** Obtenons nous les mêmes résultats avec les techniques d'échantillonnage utilisées par Halin *et al.* [1] ?

Les résultats obtenus nous ont permis de répondre à ces différentes questions. En effet pour répondre à **RQ 1.1** et **RQ 1.2**, nous avons décrit l'effort d'ingénierie nécessaire pour mettre à jour et déployer l'infrastructure de test sur la grille de calcul afin de tester toutes les configurations de JHipster. Cet effort est de **2.75 homme-mois**. Pour tester de manière exhaustive les 98154 configurations obtenues, nous avons eu besoin de **4080 heure-machine** et un espace de stockage égal à **3.5 To**.

Afin de répondre à **RQ 2.1** nous avons effectué une analyse quantitative du résultat du test exhaustif et nous avons ensuite utilisé les règles d'associations pour déterminer les interactions de Features occasionnant les Bugs afin de pouvoir les catégoriser. C'est ainsi que nous avons observé un taux de **20.59%** de configurations en échecs reparties dans **11 classes de fautes** et causées par les interactions entre 2 ou 3 Features. Une étude comparative des classes de fautes et des résultats de l'application des stratégies d'échantillonnages (PLEDGE, RANDOM, T-wise) sur les données obtenues pour les deux versions de JHipster nous ont permis de répondre à **RQ 2.2** et **RQ 2.3**. En effet, pour **RQ 2.2** nous concluons après analyse des données que les classes de fautes obtenus pour les deux versions ne sont pas causées par l'interaction des mêmes Feature car elles sont toutes différentes. Toutefois, l'interaction de la Feature *authenticationType=uaa* avec d'autres Features produit la plupart des erreurs (50.7% pour les deux versions JHipster. Nous concluons également tout

comme le décrivent Halin *et al.* que la stratégie Random est la mieux adaptée tant pour la détection de fautes que des bugs, cependant pour les échantillons de petites tailles, il serait plus intéressant d'utiliser la stratégie PLEDGE. Et donc par implication, en supposant le budget de test des développeurs de JHipster toujours invariant à ce jour (égale a 12 configurations) ces résultats nous amènent à faire la même recommandation que Halin *et al.* à savoir utiliser PLEDGE pour construire l'échantillon de 12 ou de 20 configurations à tester.

Bibliographie

[1] Axel Halin , Alexandre Nuttinck : *Sampling and Testing all configurations :The JHipster case study.* Université de Namur, Faculty of Computer Science,2017

[2] INRIA/IRISA,
`http://igrida.gforge.inria.fr`

[3] Stephen Creff : *Une modélisation de la va- riabilité multidimensionnelle pour une évolution incrémen- tale des lignes de produits.* THÈSE / UNIVERSITÉ DE RENNES 1, page 11,68

[4] K. Pohl, F. van der Linden, and A. Metzger : *Variability Management in Software Product Line Engineering. 29th International Conference on Software Engineer- ing.* Minneapolis, MN, USA, May 20-26, 2007, Companion Vol- ume, 186–187. Retrieved from http ://doi.ieeecomputersociety.org/10.1109/ ICSECOMPANION.2007.83

[5] Paul clements,Linda Northrop : *Software product lines : practices and pattern.* Addison-Wesley Professional ; 3rd edition (August 30, 2001)

[6] Felix Bachman and Paul Clements : *Variability in software product lines.* Technical report , software engineering institute, September 2005 .

[7] Klaus Pohl, Günter Böckle, Frank J. van der Linden : *Software Product Line Engineering.* Foundations, Principles and Techniques , 10 :3-540,2005

[8] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, A. Spencer Peterson : *Feature-Oriented Domain Analysis (FODA) Feasibility Study* . Software Engineering Institute Carnegie Mellon University Pittsburgh, Pennsylvania 15213

[9] Mathieu Acher, Patrick Heymans, Anthony Cleve, Jean-Luc Hainaut, Benoit Baudry : *Support for Reverse Engineering and Maintaining Feature Models* .

[10] Krzysztof Czarnecki, Chang Hwan Peter Kim : *Cardinality-Based Feature Modeling and Constraints : A Progress Report*. University of Waterloo 200 University Ave. West Waterloo, ON N2L 3G1, Canada

[11] Mikael Svahnberg, Jan Bosch : *Evolution in Software Product Lines*. University of Karlskrona/Ronneby Department of Software Engineering and Computer Science S-372 25 Ronneby, Sweden

[12] Mens Tom, Serebrenik Alexander, Cleve Anthony : *Evolving Software Systems* . Springer-Verlag Berlin Heidelberg, 2014

[13] Jan Bosch : *Design and use of software architectures : adopting and evolving a product-line approach*. Univ. of Karlskrona/Ronneby, Karlskrona, Sweden

[14] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, Patrick Heymans : *Yo Variability ! JHipster : A Playground for Web-Apps Analyses*.

[15] JHipster *Release notes :* . https ://www.jhipster.tech/releases/

[16] JHipster *Website :* . https ://www.jhipster.tech/

[17] Goetz Botterweck, Kwanwoo Lee, Steffen Thiel : *Automating Product Derivation in Softwar eProduct Line Engineering*.

[18] Nesrine Lahiani, Djamal Bennouar : *A DSL-based Approach to Product Derivation for Software Product Line* . Acta Informatica Pragensia, 2016, 5(2) :138–143

[19] Shin Yoo, Mark Harman : *Regression Testing Minimisation, Selection and Prioritisation : A Survey* . King's College London, Centre for Research on Evolution, Search and Testing, Strand, London, WC2R 2LS, UK

Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines

[20] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, Yves le Traon *Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines* . http ://www.irisa.fr/triskell/publis/2010/Perrouin010a.pdf

[21] Malte Lochau, Sebastian Oster, Ursula Goltz, Andy Schürr *Model-based Pairwise Testing for Feature Interaction Coverage in Software Product Line Engineering* . Software Quality Control, 2011

[22] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. *Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible.* In Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MODELS) ,pages 638–652. Springer, 2011

[23] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. *An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models.* In Proc. Int'l Software Product Line Conf. (SPLC) , pages 46–55. ACM, 2012.

[24] V. Chvatal. *A Greedy Heuristic for the Set-Covering Problem.* Mathematics of operations research , 4(3) :233– 235, 1979.

[25] Yu Lei,Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. *IPOG : A General Strategy for T-Way Software Testing.* In Proc. Int'l Conf. Engineering of Computer-Based Systems (ECBS) , pages 549–556. IEEE, 2007

[26] Aymeric Hervieu, Benoit Baudry, and Arnaud Gotlieb. *Pacogen : Automatic Generation of Pairwise Test Configurations From Feature Models.* In ISSRE, pages 120–129. IEEE, 2011.

[27] Sebastian Oster, Ivan Zorcic, Florian Markert, and Malte Lochau. *Moso-polite : tool support for pairwise and model-based software product line testing.* In VaMoS , pages 79–82. ACM, 2011.

[28] M.B. Cohen, M.B. Dwyer, and J. Shi. *Coverage and Adequacy in Software Product Line Testing.*. In Proceedings of the ISSTA 2006 workshop ROSATEA '06, pages 53–63, New York, NY, USA, 2006. ACM.

[29] M.B. Cohen, M.B. Dwyer, and J. Shi. *Interaction Testing of Highly Configurable Systems in the Presence of Constraints.* In Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA, pages 129–139,2007.

[30] Christopher Henard, Mike Papadakis, Gilles Perrouin, jacques Klein, Patrick Heymans and Yves Le Traon *Bypassing the Combinatorial Explosion : Using Similarity*

to Generate and Prioritize T-wise Test Suites for Large Software Product Lines. IEEE Transactions on Software Engineering, 40 :1, 2014 https ://arxiv.org/pdf/1211.5451.pdf

[31] Paul Jaccard. *Étude comparative de la distribution florale dans une portion des Alpes et des Jura.* Bulletin del la Société Vaudoise des Sciences Naturelles, 37 :547–579, 1901

[32] Iago Abal, Claus Brabrand, Andrzej Wasowski *Variability bugs in the linux kernel : A qualitative analysis.* In Proceedings of the 29th acm/ieee international conference on automated software engineering (pp. 421–432). New York, NY, USA : ACM.

[33] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, Sven Apel *A Comparison of 10 Sampling Algorithms for Configurable Systems.* In Proceedings of the 38th international conference on software engineering (icse). New York, NY : ACM Press.

[34] Xavier Devroey, Maxime Cordy,Patrick Heymans, Gilles Perrouin,Pierre-Yves Schobbens, *Towards Statistical Prioritization for Software Product Lines Testing.* In Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive (VAMOS), 2014. (pages 8, 33, 39, 51, 106) https ://arxiv.org/pdf/1310.2474.pdf

[35] INRIA/IRISA *Grille de calcul interne à l'IRISA*. http ://igrida.gforge.inria.fr

[36] Michael Hahsler, Bettina Grün, Kurt Hornik, Christian Buchta (2005, October). *arules – A computational environment for mining association rules and frequent item sets.* Journal of Statistical Software

Appendices

Annexe A

JHipster 4.8.2 : description des Technologies utilisées

Cette section présente les différentes technologies et frameworks utilisées par JHipster et montre également en les décrivant, les nouvelles fonctionnalités propres à la version de JHipster étudiée

A.1 Types d'applications générés

Nous présentons dans cette partie les différents types d'applications qu'on peut générer avec JHipster 4.8.2. En effet , l'utilisateur a le choix entre 4 types d'applications lorsqu'il désire générer une application avec JHipster. Chaque Type d'application implémente une architecture particulière et permet d'effectuer des taches bien précises.

Type d'application	Description
Application Microservice	Ce sont des applications utilisant des requêtes de type REST. Sans état, elles peuvent s'exécuter en parallèle sur le même serveur pour effectuer des taches différentes ou des services.
Application Microservice Gateway	Une application Gateway est une application qui gère les requêtes web et offre le service correspondant à la demande des applications clientes (typiquement un client AngularJS ou 4).
Application Monolithic	Ce sont de simples applications web construites avec Spring Boot côté serveur et angularJS ou Angular4 coté client.
Serveur UAA	UAA ou User Account Authentificaton, ce sont des serveurs utilisés sur des architectures de type microservice pour offrir le mode d'authentification OAuth2 pour les gateway

TABLE A.1 – Description des types d'applications

A.2 Les modes d'authentifications

La connexion à serveur nécessite très souvent une authentification, ainsi les applications générées par JHipster définissent pour chacune d'elles un mode d'authentification que ce soit pour des applications de type microservice, UAA ou monotithic. JHipster 4.8.2 définit 4 modes d'authentifications : HTTP Session , OAuth2, JWT et UAA Server

Mode d'authentification	Description
HTTP Session	Mécanisme d'authentification statefull classique basé sur le protocole HTTP
OAuth2	Mécanisme d'authentification stateless utilisant une clé secrète ou un token
JWT	Mécanisme d'authentification stateless similaire à OAuth2 mais ne requiert pas un mécanisme de persistance et peut fonctionner à la fois pour des bases de données SQL et NoSQL
UAA Server	C'est l'authentification OAuth2 définie pour les microservices

TABLE A.2 – Description des modes d'authentification

A.3 Les types de base de données

JHipster 4.8.2 supporte 9 différentes bases de données : 7 bases de données SQL et 2 NOSQL.

SQL

- MySQL
- PosgreSQL
- MariaDB
- H2(in-memory et disck-based)
- MsSQL (nouveau)
- Oracle

NOSQL

- MongoDB
- Cassandra

A.4 Les Frameworks de test

Toutes les applications générées par JHipster sont livrées avec des tests unitaires Java et JavaScript (en utilisant Karma.js). en dehors de ces tests, JHipster propose 3 frameworks aux utilisateurs, chacun d'eux se concentrant sur un aspect spécifique de l'application Web généré. Ces frameworks ne peuvent cependant pas être sélectionnés dans toutes les configurations. En effet, certaines contraintes définies dans le code du générateur permettent de n'en choisir qu'un à la fois. Il s'agit de Cucumber, Gatling et Protractor. Le tableau suivant décrit chacun de ces framework de test.

Framework de test	Description
Cucumber	Framework utilisé pour garantir la validité de certains scénarios de test
Gatling	Framework utilisé pour tester la performance d'un système
Protactor	C'est un framework de tests point à point qui permet de tester les applications AngularJS. il exécute les tests dans un navigateur comme Firefox ou Chrome. Il permet d'interagir avec les applications et de simuler le comportement d'un utilisateur.

Table A.3 – Description des Framework de test

A.5 Autres Frameworks

Il s'agit des autres frameworks intégrés par JHipster, nous retrouvons dans cette catégorie la plupart des nouvelles fonctionnalités de JHipster. Ce sont les fonctionnalités qui sont le plus susceptibles de varier d'une version à une autre de JHipster.

Frameworks	Description
Social Login	Offre la possibilité de connecter aux applications générées par JHipster via les réseaux sociaux(facebook, Google, Twitter)
ElasticSearch	C'est un moteur de recherche distribué Restfull qui accroît la capacité de recherche dans une base de données. Il est disponible pour les bases de données SQL
EhCache	C'est une cache local propre à Hibernate (JPA) permettant de maintenir en mémoire des données statiques (ou qui sont peu mis à jour) demandées par les utilisateurs. Cela accroît les performances de application.
Hazlcast	C'est une cache similaire à Ehcache mais distribuée. Elle est adapté pour les clusters et peut être utilisée pour les clusters HTTP sessions afin répliquer les session pour éviter de les perdre.
Websocket	La spécification définissant une API permettant aux pages Web d'utiliser le Protocole WebSocket pour une communication bidirectionnelle avec un hôte distant. offre une réduction énorme de la latence du réseau.
LibSass	Pré-processeur de feuilles de style pour simplifier la conception de CSS et pour traiter mise en forme conditionnelle (par exemple, si la condition est vraie, utilisez cette police, sinon utilisez celle-la).
Maven	Assure la gestion des dépendances dans un projet. C'est une des deux options pour gérer le processus complet de construction d'une application JHipster (gestion des dépendances, processus de construction, etc.).
Gradle	Semblable à maven, Gradle est réputé plus souple et plus facile à étendre.
Swagger(nouveau)	Offre des outils permettant de générer la documentation pour une API Web. Il offre également une interface permettant d'explorer et tester les différentes méthodes offertes par le service web.
ClusterHttpSession (nouveau)	Utilisé pour définir un modèle d'application avec clusters dupliquant les sessions afin de garantir la non perte d'information ou d'une session.
MessageBroker (nouveau)	Méthode permettant de distribuer les informations sur le serveur. JHipster utilise apache kafka qui permet la création de pipelines de données en temps réel transmettant de manière fiable des données entre systèmes ou applications, crée des applications de streaming en temps réel qui transforment ou réagissent aux flux de données (capture d'événement sur les réseaux sociaux : un tweet par exemple).
ServiceDiscovery (nouveau)	Utilisé pour la découverte de service dans les architectures microservice. Il permet de créer un service d'enregistrement des services afin de pouvoir les mettre à disposition des applications qui en font la demande JHipster propose deux outils (Consul développé par HashiCorp et Eureka développé par Netflix et adapté a Spring Boot).

TABLE A.4 – Description des autres frameworks

Annexe B

Description du data set JHipster 4.8.2

Nous décrivons ici la structure et le contenu du data set relatif à JHipster 4.8.2 . En effet il s'agit du contenu du fichier CSV obtenu après exécution du workflow de test sur IGRIDA. On remarque que à l'exception des nouvelles fonctionnalités , le data set reste le même que celui de la version 3.6.1 décrite par Halin *et al.* . On peut aussi remarquer que le Feature Docker possède une valeur par défaut pour toutes les configurations. Ceci s'explique par le fait que nous avons utilisé Docker pour l'exécution de toutes les configurations.

B.1. Contenu du data set

Il s'agit du contenu du fichier jhipster.csv obtenu par exécution du worflow sur IGRIDA.

JHipster Register	Docker	(...)	service Discovery Type	(...)	Compile	Log-Compile	Build	Log-Build
jhipster1	TRUE	(...)	eureka	(...)	OK		OK	
jhipster2	TRUE	(...)	consul	(...)	KO		KO	Exception make From Driver Error (SQLServerException. java :206)
(...)	(...)	(...)	(...)	(...)	(...)	(...)	(...)	(...)
jhipster98153	TRUE	(...)	false	(...)	OK		OK	
jhipster98154	TRUE	(...)	eureka	(...)	OK		KO	Exception : No instances available for uaa

TABLE B.1 – Description du fichier jhipster.csv

B.1 Description des entrées du CSV

Chaque entrée du fichier CSV représente une Feature ou une technologie utilisée par JHipster. Nous décrivons dans le tableau suivant les différentes entrées du fichier résultat ainsi que leurs éventuelles valeurs.

Entrée du CSV	Description
JHipsterRegister	Numéro du dossier dans lequel l'application Web est générée,compilée,exécutée et testée.
Docker	Valeur booléenne qui est vraie lorsque Docker est utilisé pour déployer l'application Web et false autrement
applicationType	Prenant les valeurs monolith,gateway,microservice et uaa, ce sont les types d'applications qu'offre JHipster
serviceDiscoveryType	Prenant les valeurs Eureka ou Consul, c'est l'option offerte par JHipster pour definir un service d'enregistrement et distribution de services pour les applications microservice
authenticationType	Type d'authentification ; il prend les valeurs session,jwt,uaa,oauth2
hibernateCache	Les valeurs de hibernateCache sont hazelcast,ehcache, infinispan(nouveau) ou false si le développeur ne l'active pas
clusteredHttpSession	Si elle est activé par le développeur, sa valeur est hazelcast sinon false
websocket	Si elle est activée, sa valeur est spring-websocket sinon false
MessageBroker	Sa valeur est kafka si elle est activée par le développeur, et false sinon
databaseType	Sa valeur est sql pour les bases de données SQL et mongodb ou cassandra pour le NoSQL
devDatabaseType	Pour les bases de données SQL elle a les valeurs(mysql, mariadb,postgresql ou mssql), mongodb ou cassandra pour le NoSQL et DiskBased ou inMemory pour les base de données H2.
prodDatabaseType	Même chose que DataBaseType mais sans NoSQL et H2
buildTool	Maven et Gradle sont les deux outils proposés par JHipster
searchEngine	Elasticsearch ou false si le développeur ne désire pas un search Engine dans son Application
enableSocialSignIn	Valeur Booléene true ou false si le développeur ne désire pas l'activer pour son Application
enableSwaggerCodegen	Valeur Booléene true ou false si le développeur ne désire pas l'activer pour son Application
ClientFramework	Angular4 ou AngularJS en fonction du choix du développeur et du type d'application
useSass	Valeur Booléen activé à True si activé par le développeur et False sinon
enableTranslation	Valeur Booléen activé à True si activé par le développeur et False sinon
testFrameworks	Protractor,Gatling ou Cucumber en fonction du choix du développeur
Generate	OK si la configuration génère l'application sans échec et KO sinon
Log-Generate	Trace de l'erreur de génération d'une application
Complile	OK si l'application compile sans échec et KO sinon
Log-Compile	Trace de l'erreur de compilation d'une application
Build	OK si l'application Build sans bug et KO sinon
Log-Build	Trace de l'erreur de Build d'une application

Table B.2 – Description des modes d'authentification

Annexe C

Script et algorithmes utilisés

Ces travaux étant une duplication de ceux effectués par Halin *et al.* , de plus afin d'effectuer une bonne comparaison, nous avons utilisé les mêmes scripts et les mêmes algorithmes que ces derniers. Nous reprenons donc dans cette annexe les scripts déjà présentés par Halin *et al.*[1].

C.1 Analyse quantitative et qualitative

C.1.1 Graphique présentant les proportions d'erreurs par Feature

```
data<-read.csv(file="jhipster.csv", na.strings = c("", "NA"), head=TRUE, sep=',')

data1 <- data.frame(table(data$Build,data$applicationType))
data2 <- data.frame(table(data$Build,data$authenticationType))

library(plyr)
library(scales)

names(data1)[names(data1)=="Var1"] <- "Build"
names(data1)[names(data1)=="Var2"] <- "Feature"
names(data2)[names(data2)=="Var1"] <- "Build"
names(data2)[names(data2)=="Var2"] <- "Feature"

data1 <- ddply(data1, "Feature", transform,Percentage = Freq / sum(Freq) * 100)
data2 <- ddply(data2, "Feature", transform,Percentage = Freq / sum(Freq) * 100)

data1$Feature <-
    as.data.frame(sapply(data1$Feature,gsub,pattern="uaa",replacement="uaaApp"))
data1$Feature <- unlist(data1$Feature)

data2$Feature <-
    as.data.frame(sapply(data2$Feature,gsub,pattern="uaa",replacement="uaaAuth"))
data2$Feature <- unlist(data2$Feature)

library(plyr)
dataAll <- rbind.fill(data1, data2)
print(dataAll)

library(ggplot2)

ggplot(dataAll, aes(x=Feature, y=Percentage, fill=Build,
    order=desc(Feature)),xlab='') +
geom_bar(stat="identity",colour="black") +
theme(axis.title.x=element_blank()) +
guides(fill=guide_legend(reverse=TRUE)) +
geom_text(aes(label = percent(Percentage/ 100), x =
    Feature),position=position_stack(vjust = 0.5), size = 2,colour = "black")

ggsave("bugsFeatures.pdf",height = 7, width = 9)
```

FIGURE C.1 – Script : Extraction du taux d'erreurs pour chaque Feature

C.1.2 Graphique présentant les proportions d'erreurs par classe de faute

```
1  data<-read.csv(file="jhipster.csv", na.strings = c("", "NA"), head=TRUE, sep=',')
2
3  data1 <- data.frame(table(data$Build,data$applicationType))
4  data2 <- data.frame(table(data$Build,data$authenticationType))
5
6  library(plyr)
7  library(scales)
8
9  names(data1)[names(data1)=="Var1"] <- "Build"
10 names(data1)[names(data1)=="Var2"] <- "Feature"
11 names(data2)[names(data2)=="Var1"] <- "Build"
12 names(data2)[names(data2)=="Var2"] <- "Feature"
13
14 data1 <- ddply(data1, "Feature", transform,Percentage = Freq / sum(Freq) * 100)
15 data2 <- ddply(data2, "Feature", transform,Percentage = Freq / sum(Freq) * 100)
16
17 data1$Feature <-
       as.data.frame(sapply(data1$Feature,gsub,pattern="uaa",replacement="uaaApp"))
18 data1$Feature <- unlist(data1$Feature)
19
20 data2$Feature <-
       as.data.frame(sapply(data2$Feature,gsub,pattern="uaa",replacement="uaaAuth"))
21 data2$Feature <- unlist(data2$Feature)
22
23 library(plyr)
24 dataAll <- rbind.fill(data1, data2)
25 print(dataAll)
26
27 library(ggplot2)
28
29 ggplot(dataAll, aes(x=Feature, y=Percentage, fill=Build,
       order=desc(Feature)),xlab='') +
30 geom_bar(stat="identity",colour="black") +
31 theme(axis.title.x=element_blank()) +
32 guides(fill=guide_legend(reverse=TRUE)) +
33 geom_text(aes(label = percent(Percentage/ 100), x =
       Feature),position=position_stack(vjust = 0.5), size = 2,colour = "black")
34
35 ggsave("bugsFeatures.pdf",height = 7, width = 9)
```

Figure C.2 – Script : Extraction du taux d'erreurs pour chaque classe de faute

C.1.3 Génération des Règles d'associations pour les erreurs de compilation

```
1  data<-read.csv(file="jhipster.csv", na.strings = c("", "NA"), head=TRUE, sep=',')
2
3  library(arules)
4
5  subData <- data.frame(data$Docker,data$applicationType,data$authenticationType,
6  data$hibernateCache,data$clusteredHttpSession,data$websocket,data$devDatabaseType,
7  data$prodDatabaseType,data$buildTool,data$searchEngine,data$enableSocialSignIn,
8  data$useSass,data$Build)
9
10 rules <- apriori(subData, parameter = list(minlen=2,
       maxlen=4,confidence=1,support=0.0001, target =
       'rules'),appearance=list(rhs=c('data.Compile=OK'),default='lhs'))
11 rules <- sort(rules, by = 'support')
12
13 ## redundant rules non redondunt
14 inspect(rules[!is.redundant(rules)])
```

FIGURE C.3 – Déterminer les règles d'associations pour les erreurs de compilation

C.1.4 Génération des Règles d'associations pour les erreurs de Build

```
1  data<-read.csv(file="results3.csv", na.strings = c("", "NA"), head=TRUE, sep=',')
2
3  library(arules)
4
5  subData <- data.frame(data$Docker,data$applicationType,data$authenticationType,
6  data$hibernateCache,data$clusteredHttpSession,data$websocket,data$devDatabaseType,
7  data$prodDatabaseType,data$buildTool,data$searchEngine,data$enableSocialSignIn,
8  data$useSass,data$Build)
9
10 rules <- apriori(subData, parameter = list(minlen=2,
       maxlen=4,confidence=1,support=0.0001, target =
       'rules'),appearance=list(rhs=c('data.Build=OK'),default='lhs'))
11 rules <- sort(rules, by = 'support')
12
13 ## redundant rules non redondunt
14 inspect(rules[!is.redundant(rules)])
```

FIGURE C.4 – Déterminer les règles d'associations pour les erreurs de Build

C.2 L'échantillonnage

C.2.1 Déterminer l'efficacité d'une stratégie de d'échantillonnage pour la détection des fautes

```
Sampling_Fault_Efficiency <- function(Files_Path,sample_length)
      {

        Bugs <- c(   'BUG:UaaAuthenticationWithHazelcast','BUG:UaaAuthenticationWithMicroservice',
                     'BUG:UaaAuthenticationWithNoDatabaseType','BUG:MssqlWithMonolith',
                     'BUG:MssqlWithSocialSignIn','BUG:MssqlWithServiceDiscovery','BUG:MssqlWithSession',
                     'BUG:MssqlWithOauth2','BUG:MssqlWithHibernateCache','BUG:MssqlWithWebsocket',
                     'BUG:MongodbWithGradle')

        fichiers<-dir(Files_Path,pattern = ".csv", all.files = FALSE,full.names = TRUE,
                      recursive = FALSE, ignore.case = FALSE, include.dirs = TRUE)

        n <- length(fichiers)
        m <- length(Bugs)
        Number_of_Fault_found <- numeric(n)

        for(i in 1:n){
                      Number_of_Fault<-0
                      data <- read.csv( file = fichiers[i])

                      for(j in 1:m){

                                    if( nrow(data[grep(Bugs[j],data$Bugs),])!=0)
                                       {Number_of_Fault<-Number_of_Fault+1}
                                    }

                      Number_of_Fault_found[i] <- Number_of_Fault
                     }

       moyenne<-mean(Number_of_Fault_found)
       sigma<-sd(Number_of_Fault_found)
       Efficiency<-((moyenne/sample_length)*100)
       print(moyenne)
       print(sigma)
       print(Efficiency)
    }
```

Figure C.5 – Script : Détection de fautes

C.2.2 Déterminer l'efficacité d'une stratégie de d'échantillonnage pour la détection d'erreurs

```
Sampling_Failures_Efficiency <- function(Files_Path,sample_length)
      {

            fichiers<-dir(Files_Path,pattern = ".csv", all.files = FALSE,

            full.names = TRUE, recursive = FALSE,ignore.case = FALSE, include.dirs = TRUE)

            n <- length(fichiers)

            Number_of_Failure <- numeric(n)

            for(i in 1:n){

                          data <- read.csv( file = fichiers[i])
                          Number_of_Failure[i] <- nrow(filter(data,Build=="KO"))
                          }

            moyenne<-mean(Number_of_Failure)
            sigma<-sd(Number_of_Failure)
            Efficiency<-((moyenne/sample_length)*100)
            print(sample_length)
            print(moyenne)
            print(sigma)
            print(Efficiency)
         }
```

FIGURE C.6 – Script : Détection d'erreurs

```
main <- function(args){

print("pledge8")

print("Failure : Taille de l'echantillons,moyenne, ecartType, efficacité de detection")
Sampling_Failures_Efficiency("/samples/pledge/pledge8",8)
print("Fautes: Moyenne, ecartType,efficacité de detection")
Sampling_Fault_Efficiency("/samples/pledge/pledge8",8)


print("pledge44")
print("Failure : Taille de l'echantillons,moyenne, ecartType, efficacité de detection")
Sampling_Failures_Efficiency("/samples/pledge/pledge44",44)
print("Fautes: Moyenne, ecartType,efficacité de detection")
Sampling_Fault_Efficiency("/samples/pledge/pledge44",44)
print("****************************************************************************")

print("pledge197")
print("Failure : Taille de l'echantillons,moyenne, ecartType, efficacité de detection")
Sampling_Failures_Efficiency("/samples/pledge/pledge197",197)
print("Fautes: Moyenne, ecartType,efficacité de detection")
Sampling_Fault_Efficiency("/samples/pledge/pledge197",197)
print("****************************************************************************")

print("pledge638")
print("Failure : Taille de l'echantillons,moyenne, ecartType, efficacité de detection")
Sampling_Failures_Efficiency("/samples/pledge/pledge638",638)
print("Fautes: Moyenne, ecartType,efficacité de detection")
Sampling_Fault_Efficiency("/samples/pledge/pledge638",638)
print("****************************************************************************")


print("Random8")
print("Failure : Taille de l'echantillons,moyenne, ecartType, efficacité de detection")
Sampling_Failures_Efficiency("/home/koko/Bureau/Analyse-Aout-2018/samples/Random/random8",8)
print("Fautes: Moyenne, ecartType,efficacité de detection")
Sampling_Fault_Efficiency("/home/koko/Bureau/Analyse-Aout-2018/samples/Random/random8",8)
print("****************************************************************************")
```

FIGURE C.7 – Exemple d'appel des deux fonctions