DEPARTEMENT INFORMATIQUE
DE LA FACULTE DES SCIENCES

Ahmed Kaci, Guilhèm Blanchard et Yanis Allouch

TER : Seconde analyse de la méthode VITAL et de ses métriques

HMIN201 — Travail d'Etude et de Recherche

Référent: Nicolas Hlad

2021

Introduction

B. Zhang et M. Becker introduisent leur article [ZB12] en nous rappelant de manière plus général ce que Mr Hlad nous a aussi fait remarquer sur les alternatives à FeatureIDE en contrastrant avec PureVariant et Gears.

> Successful Software Product Lines (SPLs) evolve over time.

La tendance avec les LPL [1] en particulier avec le code, c'est qu'il devient compliqué et difficile à comprendre, à maintenir, et même à utiliser après plusieurs étapes d'intégration. Refactor du code compliqué n'est pas facile.

Plus les annotations sont éparpillées, plus il est facile de les casser et d'écrire du code qui soit redondant, soit compliqué et pire que tout qu'ils soient faux ! La redondance n'est pas souhaitable non plus, car elle viole le principe DRY [2] qui est reconnu pour la bonne conception d'architecture logicielle. La complexité viole le principe KISS [3].

Pour résoudre ce problème, nous avons besoin d'outils pour analyser les annotations. La méthode VITAL introduite par B. Zhang et M. Becker, proposent 6 métriques pour cette analyse.

L'outil VITAL (Variability ImprovemenT AnaLysis), est développé pour extraire automatiquement un modèle de réflexion de variabilité à partir d'un code annoté et mené des analyses complémentaires automatiques.

Analyse

Zhang et Becker propose dans cette article une étape nommé **PLRM** [4] dans leur processus qui n'est pas pertinent d'expliciter mais nécessaire d'introduire parce qu'il les a amenées a suivre le **GQM** pour Goal-Question-Metric comme approche pour définir les métriques et la façon d'analyser le code annoté et de collecter les points de variations. En tout et pour tout : c'est la base de leur raisonnement qui les amener a construire leur modèle de réflexion de variabilité.

Dans cette seconde analyse on n'explicitera pas les métriques ni comment elles sont obtenues, ce travail a déjà été fait dans la première. Au contraire nous allons apporter des points de clarification sur certaines zones d'ombres qui ne nous étaient pas clair.

Par contre il faut rappeler qu'on peut extrapoler d'une annotation :

1. L'impact sur le code avec le nombre de bloc //#ifdef et le nombre de fichier impactés.
2. La complexité est traduite par le degré d'imbrication des blocs et leur taille.
3. Alors que le premier point permet de repéré les partie du code dont il faudra faire attention.

1. Ligne de produit logiciel, traduit de l'anglais Software Product Line (SPL).
2. Don't Repeat Yourself traduisible vers le français Ne pas se répéter soi-même.
3. Keep it simple, stupid. En français, mot à mot : « Garde ça simple, idiot ».
4. La seule signification plausible de cette acronyme qui n'est pas donné dans l'article est : PostScript Language Reference Manual (Adobe)

4. Alors le second point permet d'identifier la présence complexe ou non de point de variation.

Voici la figure 1 présentant le GQM modèle utilisé pour améliore une LPL. Refais en français, avec la traduction de *Goal* vers *But*, *Question* vers *Question* et *Metrics* vers *Métrique*, heureusement pour nous peu de chose change !

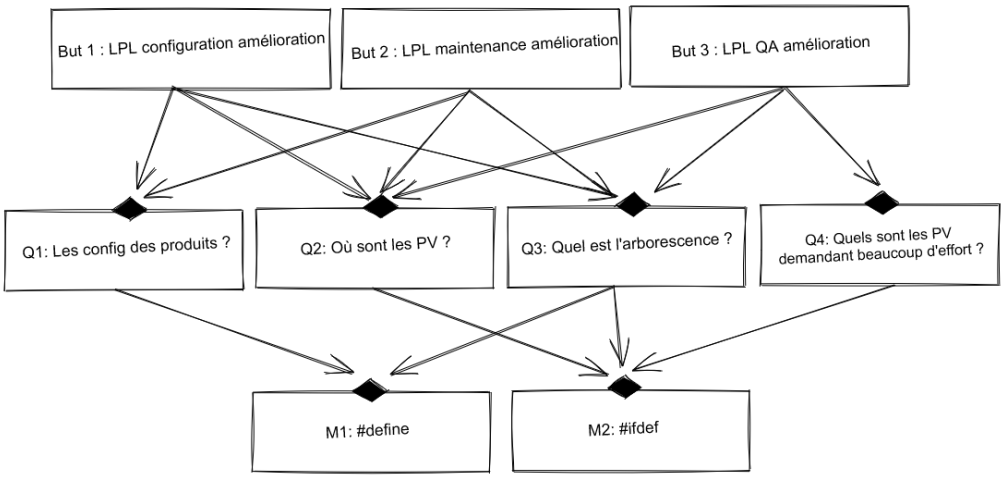

FIGURE 1 – Le GQM Modèle d'amélioration d'une LPL

Synthèse

Pour rester synthétique, voici les points nouveau de plus par rapport a la première analyse sont les suivantes :

— Parser des constantes, s'effectue via des fichiers de configuration de produit (la plus part du temps).

— Cette analyse automatisé a l'aide d'un script Python sert surtout à faire des résolutions de branchements lors de l'analyse d'annotation (la présence //#elif).

— Une solution pour ne pas crée de cycle lors du passage arborescente des annotations. Ce cycle peut provenir d'une annotation qui inclut un point de variation enfant à plusieurs niveau et endroit dans le code. On a donc un AG [5] qui ont la particularité d'être bipartie.

— Il y a une explicitation des attributs utilisé pour l'output XML utilisé en 2012.

— La citation N°8 de Kästner et al. propose une approche partiel pour pré-processer et résoudre les macros et inclusions de fichier pour les séparer du code de pré-processeur. Cette démarche s'inscrivait comme la première étape du projet TypeChef [6].

— Ils concluent leur article en précisant que l'analyse a l'étape de pré-processing est partiel et nécessite d'analyser d'autre parties du processus de LPL.

5. Acyclic Graph, remarque un AG "connecté" est appelé un arbre. Un AG "déconnecté" est appelé une foret, soit une collection d'arbre.

6. Pour rappel on a trancher que c'etait hors-sujet lors de la réunion du 12 février

Conclusion

— Les métriques de Zhang malgré la simplicité apparentes sont bien le résultat d'un raisonnement logique.

— L'exportation au format XML va sûrement être le format de prédilection pour notre propre implémentation.

— Qu'il faudra analyser les macros si elles sont présentes parce qu'elles peuvent simplifier la résolution des PV.

Références

[ZB12] Zhang, B. et Becker, M. « Code-based variability model extraction for software product line improvement ». In : *Proceedings of the 16th International Software Product Line Conference on - SPLC '12 -volume 1*. ACM Press, 2012.