

De l'adaptation de Caseine pour l'évaluation des tests des étudiants

Lydie du Bousquet, Christophe Saint-Marcel

Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, F-38000 Grenoble

1 Introduction

Nous présentons un environnement et une approche pour l'enseignement du test.

L'enseignement du test est un exercice délicat. On peut choisir d'enseigner la philosophie (e.g., "rechercher des erreurs"), les outils (e.g., JUnit, ...), des objectifs (e.g., correction fonctionnelle, performance, non régression, ...), des phases de tests (e.g. unitaire, intégration, ...), et/ou des méthodes de sélection de données.

L'évaluation du travail des étudiants se base généralement sur l'évaluation de la qualité des tests. Pour cela, on s'appuie classiquement sur une approche d'analyse mutationnelle ou d'injection de fautes. Dans les deux cas, des programmes alternatifs au programme sous test sont construits en introduisant des fautes. Les tests sont ensuite exécutés sur ces programmes fautifs. On estime que les tests sont de qualité lorsque toutes les fautes sont découvertes. Nous avons étendu l'environnement Caseine et l'approche pour l'évaluation des tests des étudiants.

2 L'environnement Caseine

Caseine¹ est une plate-forme d'apprentissage développée par l'Université Grenoble-Alpes. Elle repose sur un *Virtual Programming Lab* de Moodle (VPL). Les enseignants peuvent définir des activités pour les étudiants, puis commenter et évaluer (automatiquement ou non) les productions. En automatisant l'évaluation des productions, on fournit un retour immédiat aux étudiants, ce qui est reconnu pour influencer positivement sur la motivation et l'apprentissage [5, 6].

Pour les enseignants, Caseine offre un autre avantage : il est ouvert à toutes les universités du monde (ou presque) via la fédération d'Identités Education Recherche Edugain qui contient par exemple Renater. Par ailleurs, l'environnement est conçu pour permettre le partage des ressources. Ainsi, une large communauté (qui dépasse de loin Grenoble) contribue à l'élargissement des contenus et de la plate-forme en elle-même.

¹<https://moodle.caseine.org/>

Caseine est utilisé en particulier pour l'enseignement de la programmation. Les étudiants sont invités à compléter des programmes, puis à les évaluer en exécutant des tests définis par l'équipe pédagogique. De l'évaluation des *programmes* à l'évaluation de la qualité des *tests*, il n'y a qu'un pas... mais pas si facile à franchir. Une suite de test est de qualité lorsqu'elle met en évidence des défaillances (lorsque les tests échouent). C'est donc l'inverse de ce qui est attendu lorsqu'on évalue la qualité d'un programme.

Notre *première contribution* a consisté à implémenter ce renversement de paradigme dans Caseine (contribution technique). Cela se présente sous la forme d'un modèle de "Lab" pour Java, que l'enseignant doit adapter en définissant (a) le programme à tester et (b) les programmes fautifs qui permettront d'évaluer les tests des étudiants. Ce travail peut se faire directement en ligne ou sous Eclipse, indépendamment de la façon dont sont créés les programmes fautifs.

3 Évaluer la qualité du travail de test

Évaluer la qualité des tests. On distingue deux grandes stratégies pour produire des programmes fautifs permettant d'évaluer les tests : l'analyse mutationnelle et l'injection de fautes.

L'analyse mutationnelle s'appuie sur un ensemble d'opérateurs de mutation élémentaires, qui s'appliquent *systématiquement* sur le programme pour produire un ensemble de mutants [2]. Par exemple, un + est remplacé par un -, un < est remplacé par un > ou un ≥ (etc.). L'analyse mutationnelle présente quelques faiblesses, mais en moyenne, la communauté estime que l'approche permet d'évaluer raisonnablement la qualité des tests. C'est pourquoi, on l'utilise souvent dans des cours sur le test pour évaluer la qualité des productions des étudiants [1].

L'injection de fautes est une stratégie proche de la précédente, aussi basée sur la production de programmes erronés. Dans cette approche, les fautes choisies sont en général celles déjà identifiées dans le développement ou les promotions d'étudiants précédentes. La différence majeure entre les deux approches réside donc dans l'aspect systématisé ou non de la production de programmes erronés.

Les deux approches permettent d'évaluer la qualité des tests, mais elles ne permettent pas forcément d'évaluer la qualité de l'*application* de la méthode de tests enseignée. Quand il s'agit d'évaluer si une suite de tests couvre les instructions, les branches ou les conditions, il suffit d'utiliser des outils de couverture de code classique. Mais si l'on veut s'assurer que d'autres méthodes sont bien appliquées, il n'existe pas forcément d'outils.

Évaluer la bonne application de méthodes de tests. Notre *seconde contribution* consiste en l'adaptation du principe de l'injection de fautes. L'idée est simple. La plupart des méthodes de tests s'appuient sur un "artefact" à couvrir (élément du code ou de la spécification). Pour chaque élément à couvrir, on crée un programme erroné dont la défaillance apparaît lorsque l'exécution couvre l'artefact fixé. Par

<pre>public double foo(double a, double b){ return (a+b); }</pre> <p style="text-align: center;">(a)</p>	<pre>public double foo(double a, double b){ boolean inPartition; inPartition = (a<0) && (b<0); if (inPartition) return (a+b+1); else return (a+b); }</pre> <p style="text-align: center;">(b)</p>
--	---

Figure 1: Faute pour détecter la partie $(a < 0) \ \&\& \ (b < 0)$

exemple, pour la méthode “catégories et partitions” [3], le testeur doit définir des partitions du domaine d’entrée et choisir un test dans chaque partie. Pour vérifier que la suite de tests couvre les N parties, on crée N programmes erronés. Pour chacun, la défaillance apparaît quand le programme est exécuté avec une entrée correspondant à la partie correspondante.

Prenons l’exemple de la méthode `foo(a, b)` ayant deux paramètres réels qui retourne un réel (Fig. 1(a)). Considérons les catégories $a < 0$; $a = 0$; $a > 0$; $b < 0$; $b = 0$; $b > 0$, sans contraintes, il y a $3 * 3$ parties. Pour vérifier si la suite de tests couvre les 9 parties, on crée 9 versions fautives du programme selon le modèle donné Fig. 1(b) : une variable booléenne capture si les données correspondent à la partie considérée; le résultat est modifié si c’est le cas.

Ce principe d’injection de fautes est assez générique pour évaluer plusieurs types de méthodes de tests, comme par exemple l’utilisation du critère def-use, la couverture des états, des transitions, des paires de transitions d’un automate. A ce jour, la production de programmes fautifs est manuelle. Mais les travaux du CEA LIST sur la production d’étiquettes pour capturer différentes notions de couverture de test montrent qu’une partie du processus pourrait s’automatiser [4].

On remarquera d’ailleurs que l’on peut introduire des fautes à différents niveaux d’abstraction. Par exemple, considérons la spécification Fig. 2. Pour évaluer la couverture de transitions de la spécification (selon une relation d’équivalence de traces ici), nous avons produit deux séries de programmes erronés. Dans la première série, les fautes changent l’état d’arrivée de la transition (l’instruction (a) est modifiée). Dans la seconde série, les fautes changent le résultat retourné par la méthode (l’instruction (b) est modifiée).

Quelques exercices proposés avec trois promotions d’étudiants de l’UGA sont disponibles pour la communauté. C’est notre *troisième contribution*. D’autres exercices sont en cours de développement et seront utilisés en classe avant d’être mis à disposition.

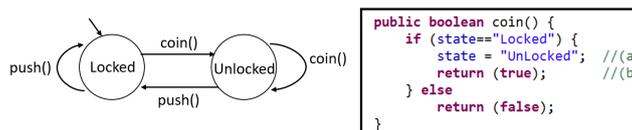


Figure 2: Spécification d’un tourniquet [7] et méthode `coin` en Java

4 Conclusion et perspectives

Cet article présente 3 contributions : (a) nous avons étendu Caseine pour l'évaluation automatisée et transparente de la qualité des tests des étudiants; (b) nous avons proposé une stratégie générique, basée sur de l'injection de fautes, pour évaluer la qualité de l'application de méthodes de test; et (c) nous avons mis à disposition des exemples pour la communauté, qui ont été utilisés avec 3 promotions d'étudiants. Les premiers retours montrent qu'ils ont apprécié l'environnement. Leurs résultats ne sont pas significativement meilleurs qu'avant, mais la comparaison est biaisée du fait des conditions d'enseignement en distanciel (contre présentiel avant).

En perspective, nous continuons à développer des exemples pour enrichir ce qui est proposé, et automatiser la production de programmes erronés pour les méthodes de test les plus simples. Nous avons aussi préparé un protocole expérimental pour mesurer l'intérêt de l'approche. Nous l'appliquerons en 2022-23.

Remerciements. Sumaiya Sultana, Al-Bashir Muhammad et Mpoki Mwaisela ont contribué à élargir le contenu des exercices dans le cadre du projet région Auvergne Rhône-Alpes RAVEN (PAI 21 007381 01) et du projet ANR PHILAE (ANR-18-CE25-0013). Le projet Caseine a été partiellement financé par l'Idex de l'Université Grenoble-Alpes (ANR-15-IDEX-0002), le LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01), et Flexi-TLV, trois Programmes Investissement d'Avenir.

References

- [1] F. Dadeau, J.-Ph. Gros, and F. Peureux. A case-based approach for introducing testing tools and principles. In *IEEE Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 429–436, 2020.
- [2] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.
- [3] S. K. Khalsa and Y. Labiche. Extending category partition's base choice criterion to better support constraints. *Journal of Software: Evolution and Process*, 30(3), 2018.
- [4] N. Kosmatov. *Combinations of Analysis Techniques for Sound and Efficient Software Verification*. Hdr, U. de Paris-Sud 11, France, 2018.
- [5] Y. Shen, Y. Wu, M. Xin, and Y. Zheng. A novel self-studying platform with its application to programming courses. In *The 9th Int. Conf. for Young Computer Scientists*, pages 2561–2566, 2008.
- [6] D. Thiébaud. Automatic Evaluation of Computer Programs Using Moodle's Virtual Programming Lab (VPL) Plug-In. *J. Comput. Sci. Coll.*, 30(6):145–151, June 2015.
- [7] Wikipedia. Automate fini. https://fr.wikipedia.org/wiki/Automate_fini, 2022.