HAL open science

Dynamic program analysis for suggesting test improvements to developers

Oscar Luis Vera-Pérez

▶ To cite this version:

Oscar Luis Vera-Pérez. Dynamic program analysis for suggesting test improvements to developers. Software Engineering [cs.SE]. Université de Rennes 1 [UR1], 2019. English. NNT : . tel-02459572

HAL Id: tel-02459572
https://hal.science/tel-02459572v1

Submitted on 29 Jan 2020

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

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

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1
COMUE UNIVERSITÉ BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

Oscar Luis VERA PÉREZ

Dynamic program analysis for suggesting test improvements to developers

Thèse présentée et soutenue à Rennes, le 17.12.2019

Unité de recherche : Inria Rennes Bretagne - Atlantique

Thèse N° :

Rapporteurs avant soutenance :

Stéphane DUCASSE	Directeur de Recherche	Inria Lille Nord Europe
Stefan WAGNER	Professeur des universités	Universität Stuttgart

Composition du Jury :

Examinateurs :	Delphine DEMANGE	Maître de Conférences	Université de Rennes 1
	Stéphane DUCASSE	Directeur de Recherche	Inria Lille Nord Europe
	Yann FONTAINE	Directeur technique	PagesJaunes
	Martin MONPERRUS	Professeur des universités	KTH Royal Institute of Technology
	François TAIANI	Professeur des universités	Université de Rennes 1
	Arie van DEURSEN	Professeur des universités	Technische Universiteit Delft
	Stefan WAGNER	Professeur des universités	Universität Stuttgart
Dir. de thèse :	Benoit BAUDRY	Professeur des universités	KTH Royal Institute of Technology

LIST OF PUBLICATIONS

- Oscar Luis Vera-Pérez, Benjamin Danglot, Martin Monperrus, and Benoit Baudry, "A Comprehensive Study of Pseudo-tested Methods", in: *Empirical Software Engineering* 24.*3* (June 2019), pp. 1195–1225, DOI: 10.1007/s10664-018-9653-2, URL: https://doi.org/10.1007/s10664-018-9653-2
- Oscar Luis Vera-Pérez, Benjamin Danglot, Martin Monperrus, and Benoit Baudry, *Suggestions on Test Suite Improvements with Automatic Infection and Propagation Analysis*, Submitted to Transactions in Software Engineering, 2019, arXiv: 1909.04770 [cs.SE]
- Oscar Luis Vera-Pérez, Martin Monperrus, and Benoit Baudry, "Descartes: A PITest Engine to Detect Pseudo-Tested Methods", in: *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*, 2018, pp. 908–911, DOI: 10.1145/3238147.3240474, URL: https://dl.acm.org/citation.cfm?doid=3238147.3240474
- Benjamin Danglot, Oscar L. Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus, and Benoit Baudry, "A snowballing literature study on test amplification", in: *Journal of Systems and Software* 157 (Nov. 2019), p. 110398, ISSN: 0164-1212, DOI: 10.1016/j.jss.2019.110398, URL: http://www.sciencedirect.com/science/article/pii/S0164121219301736
- Benjamin Danglot, Oscar L. Vera-Pérez, Benoit Baudry, and Martin Monperrus, "Automatic test improvement with DSpot: a study with ten mature open-source projects", en, in: *Empirical Software Engineering* 24.*4* (Aug. 2019), pp. 2603–2635, ISSN: 1573-7616, DOI: 10.1007/s10664-019-09692-y, URL: https://doi.org/10.1007/s10664-019-09692-y

RÉSUMÉ EN FRANÇAIS

Les logiciels ont envahi presque tous les aspects de notre vie quotidienne. Aujourd'hui, nous dépendons de plus en plus d'Internet et du Web, qui sont alimentés par des systèmes logiciels complexes et interconnectés. Nous faisons confiance aux logiciels pour protéger nos comptes bancaires et nos documents importants. Un logiciel vérifie nos passeports et nos informations de base chaque fois que nous traversons une frontière. Grâce au logiciel, nous communiquons avec d'autres personnes et partageons nos photos. Nous écoutons de la musique et regardons des films en streaming. Nous apprenons et partageons les connaissances à travers les wikis.

Le processus de développement de logiciel a évolué en poursuivant l'objectif de faire plus, en moins de temps et avec moins d'efforts. Les ingénieurs ont construit des outils pour concevoir, programmer et vérifier les logiciels. Les plates-formes modernes permettent aux développeurs de collaborer à la création de nouvelles applications et de nouveaux services et de partager leurs expériences et leurs solutions. Etant donné l'omniprésence actuelle du logiciel, il y a des conséquences lorsqu'il n'est pas correctement vérifié, qui peuvent aller d'une citation hilarante sur un écran publicitaire à la perte tragique de vies humaines.

Les tests, et plus important encore, les tests automatisés, restent au cœur des activités de développement, un moyen privilégié d'accroître la robustesse des logiciels. Mais les tests ont leur limites. "Le test de programme peut être utilisé pour montrer la présence de bogues, mais jamais pour montrer leur absence!" dit Dijsktra. Néanmoins, les méthodes formelles ne remplacent pas la nécessité d'effectuer des tests. Elles s'appuient sur des modèles mathématiques du monde réel qui peuvent faire des hypothèses irréalistes et, en tant qu'abstractions, sont différentes des machines réelles dans lesquelles les programmes sont exécutés. Les tests aident à combler cette lacune. "Méfiez-vous des bogues dans le code ci-dessus; je l'ai seulement prouvé correct, pas essayé" a écrit Knuth dans une lettre. La plupart des développeurs préfèrent les tests qui sont un bon compromis entre l'effort nécessaire et les résultats obtenus.

Le but du test est de révéler les erreurs, les défauts, les bogues dans le programme. Les tests sont conçus pour couvrir les exigences de l'application logicielle et pour mettre l'accent sur sa mise en œuvre. Les tests automatisés aident à diminuer la charge des tâches répétitives. Ils peuvent stresser le logiciel dans des conditions non prévues par les concepteurs et les développeurs, en générant de façon aléatoire de nouvelles données d'entrée. L'automatisation permet d'exécuter les tests avec un grand nombre de paramètres et de configurations, ce qui serait impossible pour l'homme. Il peut être aussi utile de repérer les régressions. Les tests automatisés sont aussi des logiciels à part entière, et ils doivent être vérifiés. Cela nous amène à nous poser une question qui sous-tend tout le travail de cette thèse : *Comment tester les tests ?*

Défis

Un test, ou un cas de test, doit fournir les données d'entrée et placer le programme dans un état spécifique pour déclencher le comportement du programme qui est jugé pertinent pour le test. Le test doit ensuite vérifier la validité de ce comportement par rapport aux propriétés attendues. Cette vérification est effectuée au moyen d'un oracle, habituellement une assertion, qui fournit le verdict. Ainsi, un test capture une interaction spécifique entre une entrée, le programme testé et un ensemble de propriétés attendues. *Tester un test* signifie évaluer la validité et la pertinence de l'interaction entre ces trois éléments, c'est-à-dire, si le test est capable de révéler des défauts du programme. Pour évaluer un test, nous devons vérifier si l'entrée est adéquate, si les comportements requis sont réellement déclenchés et si l'oracle est assez fort pour détecter les défaillances.

L'évaluation de la qualité d'un ensemble de tests, ou suite de tests, est un défi de longue date pour les ingénieurs et les chercheurs. L'amélioration de la qualité d'une suite de tests, c'est-à-dire l'amélioration de sa capacité à détecter les pannes, est encore plus difficile.

Lorsque les développeurs/testeurs conçoivent et mettent en œuvre des tests, ils ont besoin de conseils pour savoir à quel point ils testent un programme et comment améliorer leurs cas de test existants. La couverture de code est le critère d'évaluation le plus courant pour les tests, elle

indique aux développeurs quelles parties du code sont exécutées par les tests. La couverture de code informatique est efficace car elle demande peu de ressources supplémentaires pour à l'exécution régulière de la suite de tests. La plupart des languages courants disposent d'outils de calcul de la couverture et la plupart des environnements de développement modernes les supportent. Le résultat de la couverture de code est facile à comprendre, et il aide les programmeurs à découvrir ce qu'ils n'ont pas encore testé. Cependant, il se limite seulement à signaler le code qui est exécuté par la suite de tests et le code qui ne l'est pas. La couverture de code n'indique rien sur la qualité des tests et leur capacité à découvrir des bugs.

À la fin des années 70, DeMillo et ses collègues ont proposé de mettre l'accent sur les tests par mutation. [22], un critère qui subsume la couverture de code. L'idée est simple : comme les tests sont censés trouver des erreurs, cette technique implante des bugs artificiels pour créer une version défectueuse du programme, un *mutant*. Ensuite, elle vérifie si les tests existants échouent ou non lors de l'exécution du mutant. S'ils le échouent, les test détectent bien la faille implantée, dans le cas contraire, les tests devraient être inspectés et améliorés en conséquence. Les erreurs artificielles sont conçues pour imiter les petites erreurs de programmation les plus courantes en partant du principe que les programmeurs ont tendance à écrire du code proche de la vérité, et que les tests qui détectent la plupart des petites erreurs peuvent également détecter des erreurs plus importantes. L'objectif initial de DeMillo était, en fait, de fournir des conseils aux développeurs et de les aider à créer de meilleurs tests.

Cependant, et malgré des décennies de recherche scientifique sur le sujet, l'industrie adopte très lentement le test par mutation. Il y a des défis concrets à relever pour mettre la technique en pratique: (i) le processus de test par mutation est coûteux en termes de ressources informatiques; (ii) le résultat du test par mutation, fourni sous forme de score global de mutation, est difficile à comprendre et à interpréter; (iii) il est difficile de comprendre pourquoi l'interaction entre le code de test et le programme a empêché la détection d'un mutant ; (iv) il n'existe pas beaucoup de support outillé pour aider les développeurs à améliorer leurs tests pour ce type d'analyse .

Dans cette thèse, nous concevons des techniques d'analyse de programme pour relever ces défis. Ces analyses observent l'exécution de la suite de tests et identifient les parties du programme que les développeurs n'ont pas vérifiés dans leurs tests. Nos propositions sont également basées sur une forme plus légère de test par mutation, *la mutation extrême*.

Objectifs de la thèse

L'objectif principal de cette thèse est de revisiter l'objectif original de DeMillo, c'est-à-dire d'aider les développeurs à évaluer la qualité de leurs scénarios de test et à générer automatiquement des suggestions concrètes pour améliorer leurs suites de tests dans le contexte du développement logiciel moderne.

Récemment, Niedermayr et ses collègues [51] ont proposé une alternative plus légère qu'ils ont appelée mutation extrême. Au lieu de petites erreurs de programmation, ces auteurs proposent de supprimer complètement le code d'une méthode et de vérifier ensuite si les tests sont capables de détecter ce changement. D'une part, cette technique génère beaucoup moins de variantes de programme, ce qui rend l'analyse plus rapide. D'autre part, les changements non détectés sont plus faciles à comprendre au niveau de la méthode. Ces auteurs ont également découvert la présence de méthodes pseudo-testées dans des projets bien testés. C'est-à-dire une méthode dont le code peut être complètement supprimé sans qu'aucun cas de test n'échoue.

La mutation extrême travaille au niveau de la méthode, ce qui est une bonne granularité pour les développeurs pour raisonner sur le code et les tests. Dans cette thèse, nous émettons l'hypothèse que la mutation extrême et les méthodes pseudo-testées peuvent être intégrées dans les processus modernes de développement de logiciels pour fournir aux développeurs un retour d'information exploitable sur la qualité de leurs tests.

La mutation extrême et les méthodes pseudo-testées doivent fournir le cadre permettant de découvrir des problèmes de test bien localisés. Dans le cadre de ce travail, nous les étudions plus avant afin de mieux comprendre comment elles peuvent être utilisées dans la pratique. Elles sont la base sur laquelle nous construisons nos propositions.

Dans cette thèse, nous concevons et mettons en œuvre des analyses pour explorer l'interaction entre le code de l'application et la suite de tests. Ces analyses devraient aider les développeurs à comprendre où se trouvent les parties faiblement testées d'un programme (les méthodes pseudo-

testées), quels cas de test doivent être améliorés et quelles actions doivent être effectuées pour renforcer la suite de tests. Les résultats de ces analyses devraient déboucher sur des actions concrètes que les développeurs pourraient suivre pour remédier aux problèmes de test.

Notre objectif est de mettre en œuvre nos techniques d'analyse de programmes de manière à ce qu'elles puissent être intégrées dans des environnements de développement et des workflows modernes : elles doivent être compatibles avec les pratiques de construction automatique ; les développeurs doivent pouvoir les exploiter dans un environnement d'intégration continue et de test continu. Nos implémentations doivent produire une rétroaction structurée et conviviale qui peut être consultée par les humains et exploitée par d'autres outils. Nos techniques et nos résultats doivent être validés par des développements logiciels réels.

Nous divisons notre objectif principal en sous-objectifs :

I *Construire une implémentation robuste de la mutation extrême.* Afin d'obtenir un retour d'information de la part des développeurs réels, nous devons leur donner des outils qu'ils peuvent utiliser sans perturber leurs pratiques actuelles. Les développeurs devraient pouvoir utiliser ces outils dans leur espace de travail personnel et éventuellement les intégrer dans une infrastructure plus sophistiquée comme un serveur d'intégration continue. Un outil prêt pour l'industrie doit aider à atteindre les développeurs. L'insertion d'un tel outil dans un environnement de développement réel avec des pratiques de développement modernes aidera à évaluer la pertinence de la mutation extrême dans la détection des problèmes de test. Nous proposons Descartes un util qui trouve automatiquement les mutations extrêmes non detectées et les méthodes pseudo-testées.

II *Comparez les mutations extrêmes aux tests par mutation traditionnels.* Une comparaison entre les deux techniques devrait aider à déterminer dans quelle mesure la mutation extrême est la plus adaptée, et quelles sont ses principales limites. Nous utilisons un ensemble de 21 projets open-source pour mesurer la réduction en termes de mutants et de temps en cas de mutation extrême. Nous vérifions également la corrélation entre les résultats des deux approches.

III *Vérifier si des méthodes pseudo-testées et les mutations extrêmes non détectées dévoilent des résultats pertinents dans une suite de tests.* La pertinence des méthodes pseudo-testées en tant que résultats de problèmes de tests a un impact sur le type d'amélioration que nous pouvons produire à partir de leur analyse. Cette pertinence doit être établie avec l'aide des développeurs et des résultats concrets dans leur code. Pour atteindre cet objectif, nous commençons par répliquer le travail de Niedermayr et de ses collègues. Nous atténuons les menaces à la validité de leurs résultats en utilisant un outil différent et un ensemble différent de 21 sujets d'étude. Nous corroborons la présence de méthodes pseudo-testées dans des projets bien testés. Nous échangeons avec les développeurs pour évaluer la pertinence de ces résultats.

IV *Générer des suggestions concrètes d'amélioration à partir de transformations extrêmes.* Pour savoir pourquoi un mutant extrême n'est pas détecté, les développeurs doivent comprendre l'interaction entre les tests et le code de l'application. Ils devraient également améliorer leur suite de tests en créant un nouveau scénario de test, en fournissant une nouvelle entrée ou en renforçant les oracles existants. Nous proposons Reneri, un outil qui génère des suggestions concrètes pour les développeurs afin d'améliorer leur suite de tests. Reneri met en œuvre une analyse de propagation de l'infection pour découvrir dans quelle mesure les effets des mutants extrêmes se propagent à des points observables dans les cas de test. Nous corroborons, en consultant les développeurs, que ces suggestions fournissent des informations utiles pour résoudre le problème de test et même la solution exacte. Nous explorons également, si les développeurs peuvent être assistés par des outils de génération de tests à la pointe de la technologie pour résoudre les problèmes découverts par des mutations extrêmes.

Descartes un outil pour détecter les méthodes pseudo-testées

Descartes est un outil qui détecte automatiquement les méthodes pseudo-testées à l'aide de transformations extrêmes. Il s'inspire directement des travaux de Niedermayr et de ses collègues

[51]. Descartes a été conçu comme une extension de PITest pour exploiter sa maturité et ses fonctionnalités. En nous appuyant sur PITest, nous obtenons une infrastructure solide et multithread pour la découverte et l'exécution de tests unitaires, le support des outils comme Ant, Gradle et Maven, le support des bibliothèques de tests JUnit et TestNG et le support d'une communauté de chercheurs et développeurs, tous engagés pour la qualité de l'outil. Le fait de compter sur PITest a également permis une adoption plus rapide en production.

Descartes inclut des filtres spéciaux pour éviter de produire des résultats non pertinents pour les développeurs. Il inclut également des fonctions de rapports personnalisés pour présenter aux développeurs des explications en langage naturel sur les problèmes de test.

La collaboration avec nos partenaires industriels a été un élément clé dans le développement de Descartes. Des entreprises comme TellU, Atos, ActiveEon et XWiki SAS ont utilisé l'outil sur leur base de code et ont fourni un retour très intéressant et essentiel sur les problèmes et les types de résultats et de transformations qui sont les plus utiles aux développeurs. XWiki SAS a inclus des Descartes dans ses serveurs d'intégration continue pour observer automatiquement la qualité des cas de test et faire échouer la livraison lorsque cette qualité diminue. A ce jour, les contributeurs de XWiki ont fait 41 commits pour résoudre des problèmes détectés par Descartes. Ces commits ont ajouté 66 nouvelles classes de test, et ont modifié 22 et ont édité directement plus de 700 assertions dans le code. Les développeurs ont pu augmenter la couverture de code de 2% en moyenne sur chaque commit.

Extrême mutation et tests par mutation

La mutation extrême utilise des transformations de code différentes de celles traditionnellement proposées par les tests par mutation. Les mutants extrêmes éliminent en une fois tous les effets d'une méthode. Nous effectuons une comparaison quantitative entre les tests par mutation traditionnels et la mutation extrême dans 21 projets open-source. Nous observons le nombre de mutants, le temps d'exécution de l'analyse et le score de mutation. Nous montrons qu'une mutation extrême crée moins de 30% des mutants créés par la mutation traditionnelle. Dans la plupart des cas, une mutation extrême nécessite moins d'un tiers du temps requis par les tests par mutation traditionnelle. Le score calculé par les deux techniques a une corrélation positive modérée. Un projet avec un score élevé utilisant des mutants traditionnels est plus susceptible d'avoir un score élevé avec des mutants extrêmes. Nous montrons également que ces indicateurs évoluent d'un projet à l'autre en fonction de leurs particularités. Les résultats des mutations extrêmes sont à fros grain, mais ils sont en mesure de déceler des problèmes de test pertinents.

Méthodes pseudo-testées en profondeur

Nous effectuons une analyse qualitative approfondie des méthodes pseudo-testées. Dans cette analyse, nous évaluons la pertinence des méthodes pseudo-testées pour révéler les problèmes de test, en particulier les oracles faibles. Nous faisons d'abord une réplication conceptuelle du travail de Niedermayr avec un ensemble de données et un outillage différents. L'étude corrobore les premières observations de ces auteurs et atténue les menaces internes et externes à la validité de leurs résultats. L'étude trouve également des méthodes pseudo-testées dans 21 projets de Java matures. Nous confirmons que les méthodes pseudo-testées sont mal testées car le score de mutation est systématiquement plus faible que dans le reste du code.

Ces problèmes que nous avons trouvés dans cette étude ont été confirmés par les développeurs qui ont accepté les *pull requests* corrigeant les problèmes de test. Dans un échantillon de 101 méthodes pseudo-testées, les développeurs ont constaté que moins de 30% d'entre elles valaient la peine d'être testées immédiatement. Ces méthodes sont principalement impliquées dans les fonctionnalités de base et sont largement utilisées dans leur base de code. D'autre part, ils considéraient que les méthodes triviales et les méthodes d'aide n'avaient pas besoin de tests supplémentaires.

Génération de suggestions d'amélioration des tests

Nous proposons Reneri, un outil qui implémente une analyse de propagation des infections, capable de générer des suggestions pratiques qui aident les développeurs à améliorer leurs suites de tests. L'analyse examine les parties de l'état du programme qui sont affectées lorsqu'un mutant extrême est créé et que les cas de test sont exécutés. Nous montrons que dans 74% des cas, les effets de la transformation n'infectent pas l'état du programme et ne se propagent pas à un point observable. Le reste est découvert par l'amplification des capacités d'observation de la suite de tests. Les suggestions générées par l'analyse sont validées par les développeurs qui considèrent qu'elles fournissent des informations utiles, ou la solution exacte pour résoudre le problème de test.

Techniques de génération de tests contre les mutations extrêmes

Nous explorons et comparons deux techniques de détection automatisée de mutants extrêmes basées sur DSpot et EVOSUITE, deux outils pe point de génération de tests pour Java. La comparaison montre que l'approche basée sur DSpot est plus efficace car elle cible davantage les mutants extrêmes. Cependant, les solutions entièrement automatisées sont encore loin de produire un résultat totalement satisfaisant.

ABSTRACT

Automated testing is at the core of modern software development. Yet, developers struggle when it comes to the evaluation of the quality of their test cases and how to improve them. The main goal of this thesis is precisely that, to generate concrete suggestion that developers can follow to improve their test suite. We propose the use of extreme mutation, or extreme transformations, as an alternative to discover testing issues. Extreme transformations are a form of mutation testing that remove the entire logic of a method instead of making a small syntactic change in the code. As it traditional counterpart, it challenges the test suite with a transformed variant of the program to see if the test cases can detect the change. In this thesis we assess the relevance of the testing issues that extreme transformations can spot. We also propose a dynamic infection-propagation analysis to automatically derive concrete test improvement suggestions from undetected extreme transformations. Our results are validated through the interaction with actual developers. We also report the industrial adoption of parts of our results. The main contributions of this work are:

- A robust and industrial ready implementation of extreme transformations.
- A quantitative comparison between traditional mutation testing and extreme mutation considering the number of mutants, execution time of the analysis and the mutation score.
- A conceptual replication of the work of Niedermayr and colleagues who first proposed the use of extreme transformations. Our study corroborates the first observations of these authors and mitigate both internal and external threats to the validity of their results.
- An in-depth qualitative analysis of pseudo-tested methods. In this analysis we assessed the relevance of pseudo-tested methods to reveal testing issues, in particular weak test oracles.
- An infection-propagation analysis able to generate actionable suggestions that help developers improve their test suites. The analysis investigates the parts of program state that are affected when an extreme mutant is created and the test cases are executed.
- A comparison between two techniques for the automated detection of extreme mutants based on DSpot and EVOSUITE, two state-of-the-art test generation tools for Java.

Extreme transformations allow to spot concrete, relevant and easier to understand testing issues that can be leverage to produce meaningful testing improvements.

TABLE OF CONTENTS

List of publications 3

Résumé en français 4

Abstract 9

Introduction 12

1 Background and related works 19
1.1 Background . . . 19
1.1.1 Code Coverage . . . 20
1.1.2 Mutation Testing . . . 22
1.1.3 Reachability-Infection-Propagation . . . 24
1.1.4 Extreme mutation . . . 25
1.2 Related Works . . . 26
1.2.1 Mutation testing . . . 26
1.2.2 Extreme transformations . . . 27
1.2.3 Mutation testing for test generation . . . 28
1.2.4 Improving the test oracle . . . 29
1.2.5 Reachability-Infection-Propagation . . . 30
1.2.6 Tools and Industrial exploitation of mutation testing . . . 30
1.2.7 Helping testers . . . 31
1.3 Conclusion . . . 32

2 Descartes a PITest plugin for extreme transformations 33
2.1 An overview of Descartes . . . 33
2.1.1 Engine for extreme transformations . . . 34
2.1.2 Stop-methods . . . 34
2.1.3 Output of Descartes . . . 36
2.2 Descartes VS Gregor . . . 37
2.2.1 Pseudo-tested methods . . . 39
2.3 Development and adoption . . . 42
2.3.1 XWiki: an industry case study with Descartes . . . 42
2.4 Conclusion . . . 44

3 A comprehensive study of pseudo-tested methods 45
3.1 Pseudo-tested Methods . . . 46
3.1.1 Finding Pseudo-tested Methods . . . 46
3.2 Experimental Protocol . . . 47
3.2.1 Research Questions . . . 48
3.2.2 Study Subjects . . . 48
3.2.3 Metrics . . . 49
3.3 Experimental Results . . . 50
3.3.1 **RQ1**: How frequent are pseudo-tested methods? . . . 50
3.3.2 **RQ2**: Are pseudo-tested methods the weakest points in the program, with respect to the test suite? . . . 52
3.3.3 **RQ3**: Are pseudo-tested methods relevant for developers to improve the quality of the test suite? . . . 54
3.3.4 **RQ4**: Which pseudo-tested methods do developers consider worth an additional testing action? . . . 59
3.4 Threats to validity . . . 62
3.5 Conclusion . . . 63

4 Suggestions on Test Suite Improvements with Automatic Infection and Propagation Analysis **65**
4.1 Extreme transformations and test suite weaknesses . . . 66
4.1.1 Examples of undetected extreme transformations . . . 66
4.1.2 Conditions to detect extreme transformations . . . 67
4.1.3 Analyzing undetected extreme transformations . . . 67
4.2 Automatic Synthesis of Suggestions for Test Improvement . . . 68
4.2.1 Definitions . . . 68
4.2.2 Overview of the process for test improvement suggestions synthesis . . . 69
4.2.3 Infection detection . . . 70
4.2.4 Propagation detection . . . 72
4.2.5 Suggestion synthesis . . . 74
4.2.6 Implementation . . . 75
4.3 Experimental Evaluation . . . 76
4.3.1 Research questions . . . 76
4.3.2 Study subjects . . . 77
4.3.3 **RQ1**: *To what extent does the execution of an extreme transformation infect the immediate program state?* . . . 78
4.3.4 **RQ2**: *To what extent can test cases propagate the effects of extreme transformations to the top level test code?* . . . 80
4.3.5 The role of testability . . . 83
4.3.6 **RQ3**: *Are the suggestions synthesized by Reneri valuable for the developers?* 85
4.3.7 **RQ4**: *Can developers leverage test improvement tools to deal with undetected extreme transformations?* . . . 88
4.4 Threats to validity . . . 94
4.5 Conclusion . . . 95

Conclusion **97**

A List of projects used in Chapter 2 and Chapter 3 and their respective revisions **101**

B List of projects in Chapter 4 and their respective revisions **102**

C Questionnaire provided to developers in the answer of RQ3 in Chapter 4 **103**

Bibliography **105**

INTRODUCTION

Fifty years ago a man walked on the Moon. It was a great accomplishment, *one giant leap for mankind*. Software played a crucial role on this mission. It was the fourth astronaut. The Apollo program team understood very early the relevance of software. They had a glimpse of the future. They also understood that software had to be robust and reliable, and software testing was one of the most important tasks in this program. And it eventually paid off. The guidance computer was able to recover from an unexpected error and ultimately saved the mission. The Apollo program helped shape Software Engineering as a discipline on its own, and demonstrated the importance of quality and robustness of software. "Software eventually and necessarily gained the same respect as any other discipline" concluded Margaret H. Hamilton, who lead the team developing the software components for the guidance and control systems of the in-flight command and lunar modules. She, in fact, coined the term "software engineering" and her work is considered to be "the foundation for ultra-reliable software design" [8].

After five decades, software keeps going to space and has invaded almost all aspects of our daily lives. Nowadays, we depend more and more on the Internet and the Web, which are powered by complex interconnected software systems. We trust software to guard our bank accounts and important documents. Software checks our passports and background information every time we cross a border. Through software we communicate with other people and share our photos. We listen to music and watch films through streaming services. We learn and share knowledge through wikis.

The software development process has evolved by pursuing the goal of achieving more in less time and with less effort. Engineers have built tools to design, program and verify software. Modern platforms allow developers to collaborate in the creation of new applications and services and to share their experiences and solutions. Due to its omnipresence there are consequences when software is not properly verified. They may go from a hilarious "blue screen of death" in an advertising screen to the tragic loss of human lives.

Testing, and more importantly, automated testing, remains at the core of the development activities as the preferred mean to increase the robustness of software. But testing is limited. "Program testing can be used to show the presence of bugs, but never to show their absence!" said Dijsktra. But formal methods do not replace the need for testing. They rely on mathematical models of the real world that might make unrealistic assumptions and, as abstractions, are different from the real machines in which programs execute. Testing helps fill this gap. "Beware of bugs in the above code; I have only proved it correct, not tried it" wrote Knuth in a letter. Testing provides a tradeoff between effort and immediate results that is more appealing to most practicing developers.

The goal of testing is to reveal errors, faults, bugs in the program. Tests are designed to cover the requirements of the software application and to stress its implementation. Automated testing helps to alleviate the burden of repetitive tasks. It can stress software in conditions not foreseen by designers and developers by randomly generating new input data. Automation is good to run large numbers of parameters and configurations that would be impossible for humans. It can help to spot errors that reappear after being fixed. Yet, automated tests are a piece of software as well and, as such, they have to be verified. This leads to one question that underlies all the work of this thesis: *how do we test a test?*

Challenges

A test, or test case, should provide the input data and set the program in a specific state to trigger program behaviors that are deemed relevant for testing. The test must then verify the validity of this behavior with respect to expected properties. This verification is performed through an oracle, usually an assertion, which provides the verdict. Thus, one test captures a specific interaction between an input, the program under test and a set of expected properties. *Testing a test* means assessing the validity and relevance of the interplay between these three elements, that is, if the test is able to reveal program faults. To evaluate a test we must check if the input

is adequate, if the required behaviors are actually triggered and if the oracle is strong enough to spot failures.

Assessing the quality of a set of tests, or test suite, is a long standing challenge for practitioners and researchers. Improving the quality of a test suite, that is, improving its fault detection capabilities is even more challenging.

When developers/testers design and implement tests they need guidance to know how much they test a program and how to improve their existing test cases. Code coverage is the most common assessment criterion for tests, that is, let developers know which parts of the code are executed by the tests. Computing code coverage is efficient as it does not add much overhead to the regular execution of the test suite. Most mainstream languages have tools to compute coverage and most modern development environments support them out-of-the-box. The outcome of code coverage is easy to understand, it helps programmers to discover what they have not tested yet. However, it is limited to only that: signal the code that is executed by the test suite and the code that is not. Code coverage indicates nothing about the quality of the tests and their ability to discover bugs.

In the late 70's DeMillo and colleagues proposed *mutation testing* [22], a criterion that subsumes code coverage. The idea is simple: as tests are supposed to find errors, the technique plants artificial bugs to create a faulty version of the program, a *mutant*. Then the technique checks whether or not the existing tests fail when executing the mutant. If they do so, then the tests are adequate to detect the planted fault. Otherwise tests should be inspected and improved accordingly. The artificial errors are designed to mimic small traditional programming mistakes under the assumptions that programmers tend to write code close to be correct and that tests detecting most small faults can also detect bigger errors. The original goal of DeMillo was, in fact, to provide hints for developers and help them create better tests.

However, and despite the decades of scientific research on the subject, mutation testing has a languid adoption in industry. There are concrete challenges to overcome in order to put the technique in practice: (i) the mutation testing process is expensive in terms of computational resources; (ii) the outcome of mutation testing, provided as a global mutation score, is hard to understand and interpret; (iii) it is difficult to comprehend why the interaction between the test code and the program prevented the detection of a mutant; (iv) there is not much tooling support to help developers improve their tests with respect to this type of analysis.

In this thesis we design program analysis techniques to address these challenges. These analyses observe the execution of the test suite and find which parts of the program state developers have missed to verify in their tests. Our proposals are also based in a lighter form of mutation testing, *extreme mutation*.

Objectives of the thesis

The main objective of this thesis is to revisit the original goal of DeMillo, that is, to help developers *assess the quality of their test cases* and *automatically generate concrete suggestions* for them to *improve their test suites* in the context of *modern software development*.

Recently, Niedermayr and colleagues [51] proposed a lighter alternative to mutation testing which they called *extreme mutation*. Instead of small programming errors, these authors propose to completely remove the code of a method and then check if the tests are able to detect the change. For one thing, this technique generates much fewer program variants which makes the analysis faster. On the other hand, undetected changes are easier to understand at the method level. These authors also discovered the presence of *pseudo-tested methods* in well tested projects. That is, method whose code can be completely removed and no test case fails.

Extreme mutations work a the global method level, which is a good granularity for developers to reason about the code and the tests. In this thesis we hypothesize that extreme mutation and pseudo-tested methods can be integrated into modern software development processes to provide actionable feedback to developers about the quality of their tests.

Extreme mutation and pseudo-tested methods shall provide the framework to discover well localized testing issues. In this work we study them further to gain understanding about how can they be used in practice. They are the foundation in which we build our proposals.

In this thesis we design and implement a set of program analyses to explore the interplay between the application code and the test suite. These analyses aim at helping developers un-

derstand where are the weakly tested parts in a program (the pseudo-tested methods) which test cases shall be improved and what actions shall be performed to strengthen the test suite. The feedback of these analyses leads to concrete actions that developers may follow to remediate testing issues.

We aim at implementing our program analysis techniques such that they can be integrated into modern development environments and workflows: compatible with automatic build practices and exploitable in a continuous integration/continuous testing setup. Our implementations should produce structured and user-friendly feedback that can be consulted by humans and leveraged by other tools. We evaluate our concepts and tools in modular and actively maintained Java applications such as backend online services and reusable libraries. These are projects where we can execute tests on small parts of them. We aim at obtaining feedback from the main developers of these applications. Nevertheless, the concepts we develop in this thesis could be applied to other types of projects.

We break down our main objective into the following sub-objectives:

I *Build a robust implementation of extreme mutation.* In order to obtain feedback from actual developers we must give them tools they can use without disrupting their current practices. Developers should be able to use these tools in their personal working space and possibly integrate them into a more sophisticated infrastructure like a continuous integration server. An industry and production ready tool for extreme mutation shall help reach developers in this context. The insertion of such a tool into an actual development environment with modern development practices shall help assess extreme mutation in the detection of relevant testing issues. In Chapter 2 we describe Descartes, a tool to automatically find undetected extreme transformations and discover pseudo-tested methods.

II *Compare extreme mutation with traditional mutation testing.* A comparison between both techniques should help determine to what extent extreme mutation scales better, and what are its main limitations. In Section 2.2 we use a set of 21 open-source projects to measure the reduction in terms of mutants and time when using extreme mutation. We also verify the correlation between the results of both approaches.

III *Check if pseudo-tested methods and undetected extreme mutations unveil relevant findings in a test suite.* The relevance of pseudo-tested methods as findings of testing issues impacts the type of improvement we can produce from their analysis. This relevance has to be established with the help of developers and concrete results in their code. To pursue this objective we first conduct a replication of the work of Niedermayr and colleagues. We mitigate threats to the validity of their results by using a different tooling and a different set of 21 study subjects. We corroborate the presence of pseudo-tested methods in well tested projects. We interact with developers to assess if they consider these findings are relevant. Chapter 3 presents these results.

IV *Generate concrete improvement suggestions from extreme transformations.* To know why an extreme mutant is not detected, developers must understand the interaction between the tests and the application code. They should also improve their test suite by creating a new test case, provide a new input or strengthening the existing oracles. In Chapter 4 we propose Reneri, a tool that generates concrete suggestions for developers to improve their test suite. Reneri implements an infection-propagation analysis to discover to what extent the effects of extreme mutants are propagated to observable points in the test cases. We corroborate, by consulting the developers, that these suggestions provide helpful information to solve the testing issue and even the exact solution. We also explore if developers can be assisted by state-of-the-art test generation tools in the solution of the issues discovered by extreme mutation.

Contributions

The main contributions of this thesis are:

- A quantitative comparison between traditional mutation testing and extreme mutation considering the number of mutants, execution time of the analysis and the mutation score. We

show that extreme mutation creates less than 30% of the mutants created by traditional mutation testing. In most cases, extreme mutation requires less than one third of the time required by mutation testing. The score computed by both techniques have a moderate positive correlation. A project with a high score using traditional mutants is more likely to have a high score with extreme mutants. We also show that these indicators change from one project to the other depending more on their particularities. This work has been published in the Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Testing (ASE 2018) in the Tool Demonstration Track [80]. It will be described in Chapter 2.

- A conceptual replication of the work of Niedermayr with a different dataset and different tooling. The study corroborates the first observations of these authors and mitigate both internal and external threats to the validity of their results. The study also finds pseudo-tested methods in 21 mature Java projects. We confirm that pseudo-tested methods are poorly tested as the mutation score is systematically lower than in the rest of the code.
- An in-depth qualitative analysis of pseudo-tested methods. In this analysis we assessed the relevance of pseudo-tested methods to reveal testing issues, in particular weak test oracles. These issues were confirmed by developers who accepted pull requests fixing the testing issues. In a sample of 101 pseudo-tested methods, developers found less than 30% of them worth immediate testing actions. Developers would improve the tests verifying methods that support the core functionality of their product, methods that are widely used inside their projects and methods used by third parties. Developers consider worthless of additional testing actions those methods they think are trivial or support secondary functionalities *i.e.* helper methods. This work and the previous contribution have been published in the Empirical Software Engineering Journal and presented in the Journal First Track of the 41st International Conference of Software Engineering ICSE 2019. They will be presented in Chapter 3.
- An infection-propagation analysis able to generate actionable suggestions that help developers improve their test suites. The analysis determines the parts of the program state that are affected when an extreme mutant is created and the test cases are executed. We show that, in 74% of the cases, the effects of the transformation do not infect the program state nor propagate to an observable point. The rest are discovered through the amplification of the observation capabilities of the test suite. The generated suggestions are validated by developers. These developers consider that the suggestions provide helpful information, or the exact solution to solve the testing issue.
- A comparison between two state-of-the-art test generation tools in the automated detection of extreme mutants: DSpot and EvoSuite. The comparison shows that the approach based on DSpot is more effective, since it is more targeted towards extreme mutants. However fully automatic solutions are yet far from producing a completely satisfactory result. This work and the previous contribution have been submitted to the Transactions in Software Engineering Journal. They will be presented in Chapter 4.

List of scientific publications

The contributions listed above have been presented in the following publications:

- Oscar Luis Vera-Pérez, Martin Monperrus, and Benoit Baudry, “Descartes: A PITest Engine to Detect Pseudo-Tested Methods”, in: *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE ’18)*, 2018, pp. 908–911, DOI: 10.1145/3238147.3240474, URL: https://dl.acm.org/citation.cfm?doid=3238147.3240474
- Oscar Luis Vera-Pérez, Benjamin Danglot, Martin Monperrus, and Benoit Baudry, “A Comprehensive Study of Pseudo-tested Methods”, in: *Empirical Software Engineering* 24.3 (June 2019), pp. 1195–1225, DOI: 10.1007/s10664-018-9653-2, URL: https://doi.org/10.1007/s10664-018-9653-2
- Oscar Luis Vera-Pérez, Benjamin Danglot, Martin Monperrus, and Benoit Baudry, *Suggestions on Test Suite Improvements with Automatic Infection and Propagation Analysis*, Submitted to Transactions in Software Engineering, 2019, arXiv: 1909.04770 [cs.SE]

The content of these publications are adapted and included in the content of this document. Other publications made in the context of this thesis:

- Benjamin Danglot, Oscar L. Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus, and Benoit Baudry, "A snowballing literature study on test amplification", in: *Journal of Systems and Software* 157 (Nov. 2019), p. 110398, ISSN: 0164-1212, DOI: 10.1016/j.jss.2019.110398, URL: http://www.sciencedirect.com/science/article/pii/S0164121219301736
- Benjamin Danglot, Oscar L. Vera-Pérez, Benoit Baudry, and Martin Monperrus, "Automatic test improvement with DSpot: a study with ten mature open-source projects", en, in: *Empirical Software Engineering* 24.4 (Aug. 2019), pp. 2603–2635, ISSN: 1573-7616, DOI: 10.1007/s10664-019-09692-y, URL: https://doi.org/10.1007/s10664-019-09692-y

Parts of "A Snowballing ..." are included in Chapter 1.

List of tools, prototypes and open data repositories

The following tools and prototypes have been developed in the context of this thesis:

- Descartes: a PITest extension to study extreme mutations and discover pseudo-tested methods in Java projects. Descartes is a production ready tool that is also available from Maven Central.
  Repository: https://github.com/STAMP-project/pitest-descartes.
- Reneri: a tool that generates suggestions for test improvements based on an infection-propagation analysis of undetected extreme mutants.
  Repository: https://github.com/STAMP-project/reneri
- Method-inspector: A static analysis tool that classifies methods in a Maven project according to their metadata and their bytecode instructions.
  Repository: https://github.com/STAMP-project/method-inspector
- Dissector: A dynamic analysis tool that studies the interplay between a selected set of methods and the test suite in a Maven project. This tool is able to compute, among other things, the stack distance from a method to the covering test cases.
  Repository: https://github.com/STAMP-project/dissector

Both, Descartes and Reneri can be seen as main contributions of this thesis. Method-inspector and Dissector were designed and built to support the collection of data in our experiments.
All data supporting our results and additional material is publicly available for the sake of open science and reproducibility:

- Collected data and supporting scripts for the studies presented in chapters 2 and 3:
  https://ndownloader.figshare.com/files/11634542
  https://github.com/STAMP-project/descartes-experiments
- Collected data and supporting scripts for the study presented in Chapter 4:
  https://github.com/STAMP-project/descartes-amplification-experiments

Outline

The rest of this document has been organized as follows. Chapter 1 presents the background of our research and the related works to this thesis. It also presents the main concepts and the terminology that will be used in the following chapters. Chapter 2 presents Descartes, our tool to detect pseudo-tested methods with the use of extreme mutation. We discuss the main capabilities of the tool and report an industry experience of its use. We also present a comparison between traditional mutation testing and extreme mutation. Chapter 3 contains an in-depth study of pseudo-tested methods. There we perform a conceptual replication of the work of Niedermayr. We study how frequent are these methods in Java projects, we corroborate that they are the weakest points of the program with respect to testing and we explore their relevance for developers.

Chapter 4 presents an infection-propagation analysis to generate test improvement suggestions based on undetected extreme mutants. We study the main reason why these extreme mutants are not detected and which are the parts of the program state where the effects of these mutants can be detected. We present Reneri, a tool implementing this analysis and we validate our results by consulting developers. Finally we conclude this document with a recapitulation of our main contributions. We also discuss the future actions that can be taken to extend this research.

Acknowledgement

This work has been partially supported by the EU Project STAMP ICT-16-10 No.731529. The involvement of the industrial partners has been crucial for the development of this work. I owe a debt of gratitude to all the people that helped me to complete this thesis. They know who they are and I prefer to thank them personally.

CHAPTER 1

BACKGROUND AND RELATED WORKS

Developers are constantly trying to improve the quality of their code. Their goal is to concentrate on the development of the main features of their product while reducing as much as possible the occurrence of errors. As software grows in complexity so does the development process. Automated tests then become a great ally against bugs that persistently appear, disappear and sometimes, appear again.

Automated tests are usually a piece of code themselves. As such, they can be objectively improved and assessed. Developers wish to have guidance about how to consolidate their tests. They would like to know what to test in their code and how. Improving the tests directly impacts the quality of the application code.

The most used assessment criterion for automated tests is code coverage, that is, the parts of the application code that tests execute. While this criterion tells developers what they have not tested is does not say if the test cases are actually effective. The software testing literature has studied for decades stronger techniques to evaluate the quality of automated tests. Mutation testing is one of the strongest criteria. These techniques are the foundation of our work.

In this chapter we define the terminology and concepts that will be used throughout the document. In Section 1.1 we discuss the concepts related to unit testing, coverage, mutation testing and extreme mutation, a lighter alternative to mutation testing. In Section 1.2 we present a literature review of the main published works about mutation testing, how researchers propose to overcome its limitations, how it has been used to automatically improve and generate test suites and one example on how it has been used by industry.

1.1 Background

As defined in [2], *software testing* is the evaluation of a software system by observing its execution. Testing is supposed to find software faults. There are different abstraction levels at which testing can be performed. While *system testing* evaluates the architectural design to determine if the software works as a whole, *unit testing* targets the implementation bricks at the lowest level: classes, methods or functions. In this work, unless otherwise stated, when we refer to *testing* we restrict ourselves to *unit testing*.

Nowadays, it is a common practice that software, and specially open-source software, is shipped along with a considerable amount of code devoted solely to *unit testing* [87]. This code is written with the help of libraries such as JUnit [36]. These libraries provide functionalities to launch the testing process and specify concrete test cases.

These test cases are usually written as functions or methods inside specialized classes. A test case should select the appropriate input, create the required objects and trigger the required behaviors to put the unit under test in a specific state. Then, the test case must verify this state with the help of an oracle. This oracle produces a verdict that deems the state of the unit as correct or not according to the initial input. The usual oracle is an assertion, that is, a predicate that produces a truth value given the state. If the predicate evaluates to false, the test case *fails*, a *testing failure* is discovered. Our notion of a test case is formalized in Definition 1.

Definition 1. *A test case t is a method that initializes the program P in a specific state, triggers specific behaviors and specifies the expected effects for these behaviors through assertions. T is called a test suite and it is the set of all test cases t written for a program P.*

Here we consider a program P to be a set of methods. This simplified view is convenient to express the rest of the concepts in this document. When a test case t initializes a program P, it could be, in fact, a subset of P or even a single method.

Definition 1 is a simplification of what an actual test case can be in practice. The code of a test case can be much more complicated than a single method. Unit test libraries often provide

additional components that can alter the way test cases are executed and the way assertions are specified. Take for example, the `@Before` and `@After` fixtures and the `Parameterized` test classes in JUnit. As per the same definition, we understand that a set of one or more JUnit classes in a Java program is a particular case of a test suite.

Listing 1.1 shows a simple Java class. This class implements a set of objects that keeps track of the insertion operations using the `version` field (line 2). The class declares methods to check the size of the set, determine if the set contains a given object, intersect two sets and check if two given sets are equal.

In Listing 1.2 we show a test suite, in the form of a JUnit test class that contains three test cases as per Definition 1. `testAdd` (line 4) verifies that after inserting a new element, the set should be of size one. `testEquals` (line 10) verifies that two newly created sets should be equal. `testIntersection` validates that the intersection of two disjoint sets can not contain their initial elements.

In a closer look to `testAdd` we can see that lines 5 and 6 create the instance of `VersionSet` and invoke the `add` method. So these two lines set the instance of the class in the required state. Line 7 contains the assertion. The predicate checks if the size of the set is equal to one. The other two test cases have a similar structure.

1.1.1 Code Coverage

A test suite executes a portion of the program under test. It does not necessarily execute the entire program. The parts of the program that are executed by the test cases are said to be *covered* by the test suite. A *code coverage*, or *coverage*, metric assesses the size of the portion of the program executed by the test suite. As such, *line coverage* represent the lines of code covered by the test suite and *statement coverage* measures the ratio of instructions executed by all test cases. Statement coverage has been shown to better capture the effectiveness of the test suite compared to other types of coverage such as branch or block coverages [31].

Coverage metrics can be computed efficiently [73]. They usually require no more than an initial code instrumentation which does not add a great overhead in terms of execution time. There are also many high quality tools that can collect this metric for most mainstream languages. In Java, developers can use JaCoco[1], Cobertura[2], OpenClover[3] among others. These tools are often included in popular Integrated Development Environments, such as Visual Studio and IntelliJ. Popular Continuous Integration Servers like Jenkins or Travis also support code coverage computation.

Listing 1.1 also shows the line coverage that `VersionedSetTest` achieves over `VersionedSet`. Lines highlighted in green are those executed by the test suite. In turn, lines highlighted in red are not executed by the test suite. One can see that the test class is able to cover a large portion of the class under test. For example, all lines of `intersect` except lines 48 and 53 are executed by the test suite.

One test case can invoke, directly or not, several methods of the program under test. In Listing 1.2 `testAdd`, triggers the execution of the constructor of the class and the `add` method, who also invoke `contains` and `incrementVersion`. We say that all these methods are *covered* by `testAdd`. Definition 2 formalizes this concept.

Definition 2. *A method $m \in P$ is said to be covered if there exists at least one test case $t \in T$ that triggers the execution of at least one statement of the body of m.*

[1]https://www.jacoco.org/jacoco/
[2]https://cobertura.github.io/cobertura/
[3]http://openclover.org/

```java
public class VersionedSet {
  private long version = 0;
  private ArrayList elements = new
      ArrayList();

  public void add(Object item) {
    if (contains(item))
      return;
    elements.add(item);
    incrementVersion();
  }

  private void incrementVersion() {
    version++;
  }

  protected long getVersion() {
    return version;
  }

  public int size() {
    return elements.size();
  }

  public boolean isEmpty() {
    return size() == 0;
  }

  public boolean contains(Object item) {
    return elements.contains(item);
  }

  @Override
  public boolean equals(Object otr) {
    if(!(otr instanceof VersionedSet))
      return false;
    VersionedSet otrSet = (VersionedSet)otr;
    if(otrSet.size() != size())
      return false;
    for (Object item : elements) {
      if(!otrSet.contains(item))
        return false;
    }
    return true;
  }

  public VersionedSet
      intersect(VersionedSet otr) {
    if (isEmpty() || otr.isEmpty()) {
      return new VersionedSet();
    }
    VersionedSet result = new
        VersionedSet();
    for(Object item : elements) {
      if(otr.contains(item)) {
        result.add(item);
      }
    }
    result.version = 0;
    return result;
  }
}
```

Listing 1.1: A class under test

```java
public class VersionedSetTest {

  @Test
  public void testAdd() {
    VersionedSet list = new
        VersionedSet();
    list.add(1);
    assertEquals(1, list.size());
  }
  @Test
  public void testEquals() {
    VersionedSet one = new VersionedSet();
    VersionedSet two = new VersionedSet();
    assertTrue(one.equals(two));
  }
  @Test
  public void testIntersection() {
    VersionedSet one = new VersionedSet();
    one.add(1);
    VersionedSet two = new VersionedSet();
    two.add(2);
    VersionedSet result =
        one.intersect(two);
    assertFalse(result.contains(1));
    assertFalse(result.contains(2));
  }
}
```

Listing 1.2: Test class verifying `VersionedSet`

```java
@Test
public void testAdd() {
    VersionedSet list = new VersionedSet();
    list.add(1);
    list.size();
}
```

Listing 1.3: `testAdd` with no assertion

By extension, a method is *partially covered* if at least one statement is not executed by any test case and *fully covered* otherwise. `add` is an example of a fully covered method, while `intersect` is only *partially covered*.

Code coverage is useful to determine the parts of the code that are not being tested. For example, since line 48 is not being covered one can deduce that no test case considers the intersection with an empty set. A new test case is then required to cope with this situation.

Coverage metrics are often used as a synonym of test quality [31]. The higher the coverage, the better the test suite is considered to be. However, a high coverage ratio does not necessarily mean that the test suite is effective. Besides, achieving in practice a 100% of code coverage is unrealistic as it demands great effort from developers and might produce not useful test cases.

It can be expected that, in a well tested codebase, the test suite achieves a high coverage but the opposite is not true in general. As a rather extreme example, one can consider to remove all assertions in Listing 1.2. The new test cases would be able to achieve the same line coverage but they would not be checking the behavior of the class.

For instance, removing the assertion of `testAdd` (line 7 in Listing 1.2) produces the test case shown in Listing 1.3. This new test case triggers the required behaviors and have the same coverage as the original test case. However, this test case is not able to detect changes in the behavior of the code. If we remove the body of `add` (line 5) and execute the test case no failure will be reported. The test case is not able to spot the bug. In this sense we say that the test case does not check the behavior of the method. This example manifests that only considering the coverage of the test suite is not enough to establish its quality.

Mutation testing, introduced in the late 70's by DeMillo and colleagues[22, 23] is a much better assessment for automated test cases. The technique introduces artificial faults in the program and then checks if the tests are able to fail when executing the faulty program. Decades of research have been dedicated to mutation testing, however, it is not widely used by industry practitioners. Mutation testing is a very expensive analysis and its outcome is hard to understand most of the times. The majority of research focus the attention in making the process more efficient or improving the mutation score, that is, the ratio of artificial faults a test suite can find.

The original goal of mutation testing was not to compute a score, but to provide hints for programmers on how to select their test data and handle the artificial fault, which are proxies of potential programming errors. In our work we have a similar goal: give developers actionable and concrete hints on how to improve their unit test cases. Niedermayr *et al* [51] introduced the concept of extreme mutation and pseudo-tested methods. Their technique seems to be able to discover concrete and well located testing issues that developers should solve to improve their test suites.

In the following sections we explain both, mutation testing and extreme mutation.

1.1.2 Mutation Testing

Mutation testing, *mutation analysis* or originally *program mutation* was introduced by DeMillo *et al* [22] as an automated procedure to evaluate test suites.

The technique is based on two main assumptions:

- Programmers create programs that are close to being correct. That is, competent programmers make small mistakes. (*The competent programmer hypothesis*)
- A test suite that detects all simple faults can detect most complex faults. That is, complex errors are coupled to simple errors. (*The coupling effect*)

So, mutation testing plants small artificial faults in the program under test in the form of common programming errors. Then, it verifies if the test suite is able to detect the planted changes.

Each program variant created after introducing a fault is called a *mutant*. A mutant is said to be *killed* if the execution of at least one test case fails, that is, if it is detected by the current test suite. Otherwise it is said to be a *live* mutant.

The model of faults that are introduced in a program are usually called *mutation operators*. These operators perform small syntactical changes in the source code. The changes are designed to mimic common mistakes that programmers tend to make. Typical mutation operators would change a comparison operator by other, change an arithmetic operator, slightly alter the result of a method or change a constant value.

Data: P, T, Ops
Result: $score, live, killed$
1 $M \leftarrow \text{generateMutants}(P, T, Ops)$
2 **foreach** $m \in M$ **do**
3   **if** $\text{run}(T, m)$ **then**
4         $killed \leftarrow m$
5   **end**
6   **else**
7         $live \leftarrow m$
8   **end**
9 **end**
10 $score \leftarrow \frac{|killed|}{|M|}$
11 **return** $score, live$

Algorithm 1: Mutation analysis

```java
// Mutant 1
public class VersionedSet {
    ...
    protected long getVersion() {
        return version + 1;
    }
    ...
}

// Mutant 2
public class VersionedSet {
    ...
    private void contains(Object item) {
        return !element.contains(item);
    }
    ...
}

// Mutant 3
public class VersionedSet {
    ...
    public boolean incrementVersion() {
        version--;
    }
    ...
}
```

Listing 1.4: Two examples of mutants created from `VersionedSet`

Listing 1.4 shows three examples of mutants that can be created for the `VersionedSet` class in Listing 1.1.

The first mutant (line 5) adds one to the result of `getVersion`. Line 5 is not executed by the test suite, so this mutant is not *covered* and therefore not detected. The second mutant (line 14) negates the condition in the return value of `contains`. This mutant is covered by `testAdd` and it is killed by the same test case. `contains` is expected to return false in line 6 of `add`. The mutant makes the method return true, the element is not inserted and the assertion fails as the set is empty. The third mutant in Listing 1.4 (line 23) replaces the increment operator by a decrement operator. As the value of `version` is not assessed by any test case the mutant is not killed and *survives* the analysis.

Algorithm 1 shows the general implementation of the mutation analysis. Every mutant is analyzed in isolation. The result of one mutant is not expected to affect the outcome of another. This is not always true in practice. Some tests may modify external data or files that might affect the initial conditions of other tests or even the same test when executed with a different mutant.

The mutation analysis produces, as a raw output, the list of live and killed mutants. Today these lists are usually turned into a *mutation score* (line 11): the ratio of detected or *killed* mutants to the

total number of mutants created for a program. Yet, the score is a single metric that summarizes the entire process so it does not provide detailed information about the test suite. Its value is highly dependent on the mutation operators. Different sets of operators are expected to produce different scores.

Each live mutant may enclose a testing issue or flaw. They may uncover behaviors that are not being assessed by the test suite or scenarios that testers might have missed. Live mutants could be semantically equivalent to the original code. Determining if the mutant is equivalent to the original program is undecidable in the general case. Testers/developers should analyze each live mutant. Developers must discard those that are equivalent. They also should comprehend why the existing test suite is not able to detect a given live mutant and improve the test cases accordingly.

Mutation testing does not have a wide industrial adoption, despite being a simple and effective idea. The related literature [50, 47] explains this scarce use beyond academic circles with the following reasons:

- The cost of the analysis: The number of mutants that can be created is huge even for simple programs. Each mutant must be created and challenged against the test suite. These factors make the analysis very expensive in computational terms.
- The presence of equivalent mutants: The mutation operators may create program variants which are equivalent to the original code and thus indistinguishable from live mutants. Each live mutant must be manually analyzed to see if it is equivalent to the original code. This is a hard and highly time consuming task. However, Coles [12], has suggested that equivalent mutants can unveil parts of code requiring refactoring.
- The lack of integrated and production ready tools. Even when there are several tools for mutation testing, most of them are created for academic purposes. Only a few alternatives like PITest [13] are used in industrial contexts.

We can add yet another element to this list: *the lack of a clear methodology on how to use mutation testing*. In our own experience with industrial partners [77] we have observed that the mutation score evolves in a different way than code coverage, which is confusing for developers. As a global metric the score is hard to interpret and produce a direct test improvement. Developers find more value in discovering why a mutant is not killed by the test suite. But, as other authors have also noticed [26], this is a hard task that requires a deep insight on the application code and the test cases.

Most of the research related to mutation testing is directed to:

- Determine if the mutation operators are representative of actual programming errors.
- Overcome the limitations of the technique. A considerable number of works explore de use of reduced sets of mutants, selected mutation operators, strategies for faster test execution and proposals to avoid the creation of equivalent mutants.
- Use the mutation score as a fitness function for automatic test generation.

Little effort has been devoted to the help testers understand the interplay between the test suite and live mutants and how to leverage this information to create new test cases or improve the already existing test suite. Surprisingly, this was the actual goal of the seminal work of DeMillo and Lipton: " help developers improve their test cases through the hints provided by live mutants".

The main objective of this thesis is to investigate the original intent of DeMillo's work in the context of modern software development practices and leveraging extreme mutation to discover testing issues.

1.1.3 Reachability-Infection-Propagation

A fault, real or artificial as in mutation testing, can only be discovered if the following three conditions are met at the same time: (i) the fault is executed/*reached*; (ii) after the execution of the fault, the program state changes/*becomes infected*, and (iii) the program infection *propagates* to an observable point or the output.

These three conditions conform the *Reachability-Infection-Propagation* (RIP) model for fault detection [48, 49, 24].

Li and Offutt [45] evolve the RIP model into *Reachability Infection Propagation Revealability*, (*RIPR*) to include the notion that the oracle in automated tests must *reveal* failures.

We can replay the analysis of the mutants Listing 1.4 using the conditions of the RIP model. The first mutant (line 5) is not *reached* by any test case. The second mutant (line 14) is *reached* by `testAdd`, its effects *propagate* to the code of the same test case and one assertion *reveals* the infection. In turn, the test suite reaches the third mutant (line 23) but the infection does not propagate to an observable point and therefore can not be revealed.

The RIP model provides a methodological framework to study why a fault or mutant is not revealed by a test suite. In fact, the work of Voas [81] leverages a similar notion, the *Propagation-Infection-Execution Analysis* (PIE). This proposal performs a dynamic analysis to determine the likelihood for faults to be detected according to their location in the code.

In this work we leverage this framework to analyze an alternative approach to mutation testing: *extreme mutation*.

1.1.4 Extreme mutation

Niedermayr *et al* [51, 52] introduced in 2016 the concept of extreme mutation. In their own words, this is a *lightweight* alternative to traditional mutation testing.

They propose to use a different kind of mutation operators that work at the method level. If the method is void, then all instructions are removed. If the method returns a value, then the entire body is replaced by a single return instruction with a predefined constant value.

```
1  // Extreme Mutant 1
2  public class VersionedSet {
3   ...
4   public void add(Object item) { }
5   ...
6  }
7
8  // Extreme Mutant 2
9  public class VersionedSet {
10  ...
11  private void equals(Object otr) {
12   return true;
13  }
14  ...
15 }
16
17 // Extreme Mutant 3
18 public class VersionedSet {
19  ...
20  private void equals(Object otr) {
21   return false;
22  }
23  ...
24 }
25
26 // Extreme Mutant 4
27 public class VersionedSet {
28  ...
29  public boolean incrementVersion() { }
30  }
31  ...
32 }
```

Listing 1.5: Four examples of extreme mutants created from `VersionedSet`

Listing 1.5 shows four extreme mutants created from `VersionedSet`. The first (line 4) and fourth (line 29) extreme mutants are created in void methods. The second and third mutants are created both in the `equals` method. The first extreme mutant is detected by `testAdd`. As no

object is actually inserted in the set, the collection remains empty and the assertion of the test case fails. The third extreme mutant is also killed, as the `equals` method is expected to return true in `testEquals`. The second and fourth extreme mutants are not detected by any test case.

According to Niedermayr *et al*, extreme mutation has two main advantages: (i) it creates less mutants which makes the analysis faster and (ii) most equivalent mutants can be detected by analyzing the code.

To our view, extreme mutation has two additional advantages. The reduced number of mutants points to a much smaller and therefore more targeted set of potential testing issues that developers can handle. As they work at the method level, developers can understand their effects more easily compared to traditional mutants.

Niedermayr *et al* use extreme mutants to discover *pseudo-tested methods*. These are methods where the test suite did not detect any extreme mutant. In Listing 1.5 `incrementVersion` is pseudo-tested, as the only extreme mutant it has, was not detected. `add` and `equals` are not pseudo-tested. The only extreme mutant of `add` is detected and the `return false` mutant for `equals` was detected as well.

Pseudo-tested methods are arguably the worst tested methods in a project. Their effects can be suppressed at once, yet no test case is able to notice the change. As a very interesting fact, Niedermayr and colleagues found pseudo-tested methods in all the projects they studied.

Extreme mutation does not follow any fault model. We can say that it violates the competent programmer assumption. As such, to further differentiate the two techniques and avoid confusion, we will refer to extreme mutation as *extreme transformations*. We say a extreme transformation is *detected* when the extreme mutant is killed otherwise it is *not detected*.

1.2 Related Works

Our work is directly inspired by the seminal work of DeMillo and colleagues [22] and the proposal of Niedermayr and colleagues [51]. It is also related to several fields of the software testing literature: dynamic analysis of test execution, test improvement and mutation-based test generation. In the following sections we present a review of the most representative works in the area.

1.2.1 Mutation testing

In 1978, DeMillo and colleagues proposed a seminal piece of work in the area of software testing; "Hints on Test Data Selection: Help for the Practicing Programmer" [22]. This work is mostly known for introducing mutation analysis as a novel test assessment criterion, stronger than code coverage, and for the whole literature that has followed. Meanwhile, our work is more inspired by the paper's intention to provide "hints on test data selection". In particular, the following concluding remark is a key motivation for our work: mutation analysis "yields advice" to be used in generating test data for similar programs". While this quote dates from four decades back, there has been very little work that pursued this idea of "advice" extraction from mutation analysis. Our work explores this opportunity further, in the context of extreme transformations.

Traditional mutation testing has been evaluated against real faults in several occasions [17, 3, 37]. The evidence presented supports that mutation testing is able to create effective program transformations under the assumption that programming errors are generally small and complex faults can be detected by tests which also detect simpler issues.

Gopinath *et al* [32, 33] study the limits of the two assumptions on which mutation testing is based. These authors investigated a total of 240000 bug fixes across 5000 programs written in four different programming languages [32]. They concluded that a significant number of changes are larger than the ones created by traditional mutation operators, which suggests that, in this sense, real faults may be different from mutants. They also observe that there are differences in the patterns of changes among different programming languages and that the mutation analysis also exhibits differences in this aspect. This fact was also observed in practice by Petrovic and Ivankovic [59]. In a later work [33] the same authors state that the understanding of the coupling effect is yet incomplete. They propose the *composite fault hypothesis* which claims that tests detecting a fault in isolation have a high chance to detect the fault when it occurs in combination with others.

A large body of works have been dedicated to make mutation analysis more efficient. Untch [75] divides these works in three main strategies:

- *do fewer*: these approaches "try to run fewer mutated programs without incurring intolerable loss in effectiveness".
- *do smarter*: which "distribute the computational expense over several machines or factor the expense over several executions by retaining state information between runs".
- *do faster*: these "try to generate and run mutant programs more quickly".

Works following the *do faster* and *do smarter* strategy propose to integrate mutation operators in the compilation process to speed up mutation creation [21], or propose a cloud infrastructure to distribute the analysis and make it faster [10, 64].

A notable set of works has been devoted to reduce the number of mutants in the analysis, (the *do fewer* approach). Some authors propose to randomly sample the mutants to be used [9, 1, 85]. Other authors propose to use only a subset of mutation operators [53]. Another group of works explore the trade-offs of mutant sampling and operator-based selection [84, 89, 88]. The use of higher order mutants, that combine several first order traditional mutants, has been explored as a way to reduce the execution time [62] and deal with equivalent mutants [40].

Kurtz *et al* [43] state that the mutation score is affected by the presence of equivalent mutants and redundant mutants, that is, mutants that are killed by the same test cases that detect others. A mutation score that eliminates such redundancies would be a better assessment for the detection capabilities of a test suite. These authors also show [42] that mutation selection approaches do not take this into account and therefore their evaluation is imprecise.

Untch [75] proposed to use statement deletion operators. It shows a drastic decrease on the number of mutants while maintaining the accuracy. The idea was expanded by Deng *et al* [25] and Delamaro *et al* [19] to additional programming languages and the deletion of blocks, variables, operators and constants. These approaches create much fewer mutants than the traditional mutation operators with decreases below 80%. Durelli et al. [26] observe that these statement deletion mutants are as time-consuming as the traditional operators when developers try to determine their equivalence to the original code.

Pizzoleto *et al* [61] present a more up-to-date survey on mutation cost reduction techniques. These authors provide a comprehensive list of cost reduction goals, techniques and reduction metrics used in the literature. Papadakis *et al* [56] also present an extensive review of recent advances in the research related to mutation testing.

1.2.2 Extreme transformations

Niedermayr *et al* [51] presented a mutation testing alternative that uses extreme transformations originally coined as *extreme mutation*. These transformations work at the method level, contrary to traditional mutants, that modify single instructions or blocks.

Extreme transformations completely alter the body of a method. If the method is void, all instructions are removed. If the method returns a value, then all instructions are replaced by a single return statement with a predefined constant value. In this way, all side effects of the method are removed and the result value would be the same for each invocation triggered by the test suite. More than one transformation can be applied to a non-void method if the process considers different constants of the same type. A method is said to be pseudo-tested when none of the extreme transformations created on its body are detected by the test suite.

These transformations are less prone to create a program variant equivalent to the original source code. It is possible to detect methods whose structure is as simple as the ones created by the transformations with a simple static analysis. Also, the number of mutants that are created for an entire program is much lower than what traditional mutation testing would create. These two aspects directly account for two limitations of mutation testing. The effects of these transformations should be easier to understand by developers, as the modifications affect the entire method.

Niedermayr and colleagues explored 14 projects as study subjects. They were able to find pseudo-tested methods in all these projects with ratios ranging from 6% to 53%. They also show that pseudo-tested methods tend to be related with test cases that execute a large portion of the entire system and that a significant number of these methods contains relevant functionalities.

The ratio of pseudo-tested methods allowed these authors to conclude that code coverage is only an indicator of the effectiveness of the test suite when it comes to unit test cases.

In a later work, Niedermayr and Wagner [52] showed a correlation between the stack distance from the test code to the pseudo-tested method and the test effectiveness in 21 open-source projects. Methods with a higher stack distance from test cases are more likely to be pseudo-tested and therefore not well tested. These authors show that a classifier, trained with features such as the stack distance, line and branch coverage, number of covering test cases, number of invocations and others can predict whether a method is effectively tested or not with a 92.9% precision and 93.4% recall.

In a sense, extreme transformations are a form of higher order mutation operators and a special case of instruction removal operators. The do not follow, however, any fault model, unless considering the extreme case where a developer did not implement a method.

The concepts introduced by Niedermayr and colleagues are intriguing. In this work we further investigate extreme transformations and pseudo-tested methods and explore how developers can use them to improve a test suite.

1.2.3 Mutation testing for test generation

Mutation testing evaluates the fault detection capabilities of the test suite. The outcome of this analysis and, in particular, the mutation score, has been used to guide the automatic generation of test cases and test inputs able to reveal regression faults in the system under test. In this section we review some of these works.

DeMillo and Offutt [24] generate test data based on mutation testing. They derive the conditions/constraints under which the execution of a mutant produces a different state than the original code and then generate test inputs with a constraint solver.

Another group of works have proposed search-based approaches. Baudry *et al* [6, 7] improve the mutation score of an existing test suite by generating variants of existing tests through the application of specific transformations of the test cases. They iteratively run these transformations, and propose an adaptation of genetic algorithms (GA), called a bacteriological algorithm (BA), to guide the search for test cases that kill more mutants. The results demonstrate the ability of search-based approaches to significantly increase the mutation score of a test suite. They evaluated their proposal on 2 .NET classes. The evaluation shows promising results, however the result have little external validity since only 2 classes are considered.

Pacheco and Ernst [55] proposed Randoop, a tool to randomly generate regression tests. The tool creates method sequences and checks the execution of these sequences against a given set of contracts. Those executions that do not violate the contract are given back as regression tests.

Fraser and Zeller [29] present μTest, a genetic algorithm implementation to generate unit tests based on mutation analysis. The fitness function they use combines notions of how close is the test case to execute the mutated statement and whether the execution of the mutant leads to an infection of the program state and a subsequent propagation. The tool observes the objects created in the test cases to extract field values and the results of method invocations. The observed values are used to generate a reduced set of assertions, able to tell the original code apart from its mutants. The idea is further explored by Fraser and Arcuri and implemented a later tool named EVOSUITE [27].

Using two open source programs as study subjects, Smith and Williams [69] empirically evaluate the usefulness of mutation analysis in improving an existing test suite. They execute the existing test cases against a set of generated mutants. Then, for each mutant that is not killed, a new test case is written with a single intention: kill this mutant and only this one. Their results reveal that a majority of mutation operators are useful for producing new tests, and the focused effort on increasing mutation score leads to an increase in line and branch coverage. The same authors conduct another study [70] that confirms this finding. It also emphasizes the importance of choosing the appropriate mutation tool and operators to guide the production of new test cases.

Rojas *et al* [63] have investigated several seeding strategies for the test generation tool EVOSUITE. Traditionally, EVOSUITE generates unit test cases from scratch. In this context, seeding consists in feeding EVOSUITE with initial material to start the automatic generation process. The authors evaluate different sources for the seeds: constants in the program, dynamic values, concrete types and existing test cases. The experiments with 28 projects from the Apache Commons repository show a 2% improvement of code coverage, on average, compared to a generation from

scratch. The evaluation based on Apache artifacts is stronger than most related work, because Apache artifacts are known to be complex and well tested.

These works use traditional mutation testing as an objective function to guide automatic test generation. As a consequence, the set of generated tests is expected to have good fault detection capabilities. However, none of these works validate the relevance of mutants for developers nor they assess the utility of the generated tests.

In our work we leverage extreme transformations. Our goal is not to automatically generate test cases but to provide guidance to developers in the form of suggestions. The suggestions also seek to improve the test suite in terms of fault detection. Although, we do evaluate how automatic test generation tools perform when targeting extreme transformations.

1.2.4 Improving the test oracle

The automation of the test oracle is of great relevance for software testing. In this section we review a selection of works oriented to generate or improve test oracles.

Schuler and Zeller [67] introduce the concept of checked coverage, as the percentage of program statements that are executed by the test suite and whose effects are also checked in the oracles. They compare this metric to code coverage and mutation testing with respect to their ability at assessing oracle decay. They perform manual checks on seven real software projects and conclude that checked coverage is more realistic than the usual coverage.

Jahangirova et al. [35] propose a technique for assessing and improving test oracles. They use mutation testing to increase the fault detection capabilities of the oracles and automated unit test generation to reduce the number of correct executions rejected by the assertions. Their approach is shown to be effective in five real software projects. The fault detection ratio, approximated with the mutation score, is increased by 48.6% in average.

Pacheco and Ernst implemented Eclat [54], which aims to help the tester in the creation of new test inputs with constructed oracles. Eclat first uses the execution of some available correct runs to infer an operational model of the software. Eclat then uses this model and a classification-guided technique to generate new test inputs. Next, the tool reduces the inputs by selecting only those that are most likely to reveal faults. Finally, Eclat adds an oracle for each remaining test input from the operational model.

Fraser and Zeller [28] propose an approach to generate parametrized unit tests containing symbolic pre- and post-conditions. Taking concrete inputs and results as inputs, the technique uses test generation and mutation to systematically generalize pre- and post-conditions. Evaluation results on five open source libraries show that the approach can successfully generalize a concrete test to a parameterized unit test, which is more general and expressive, needs fewer computation steps, and achieves a higher code coverage than the original concrete test.

Xie [86] amplifies object-oriented unit tests by adding assertions on the state of the receiver object, the returned value of the tested method and the state of parameters. The approach, named *Orstra*, instruments the application code and runs the test suite to collect state of objects. Orstra generates assertions y calling observer methods, that is, non-void methods. The author evaluates the tool with 11 Java classes different in number of methods and lines of code. The evaluation uses tests generated by two external tools. The results show that Orstra can effectively improve the fault-detection capabilities of the original tests.

Daniel *et al* developed ReAssert [16] a tool that suggests test repairs to developers. The tool tries to change the behavior of failing test cases to make them pass while trying to maintain the original test code as much as possible and retaining the test's regression detection capabilities. The resulting test case is proposed to the developers as a potential fix. The tool records the values of failing assertions and leverages the information in the failure exception to locate the code to repair and select a repair strategy from a predefined set. This is repeated until no assertion fails. The tool is evaluated in six open-source projects and was able to produce fixes from 25% to 100% of failing tests for all their study subjects. The authors also carried out a usability study, where developers were asked to write failing tests. The tool was able to solve 98% of them.

Staats *et al* [72] propose an automated process to support the creation of test oracles. These authors are interested in the creation of test oracles defined as concrete values the system under test is expected to produce. Their goal is not to fully automate the oracle creation, but to guide the tester in the selection of variables and outputs for which the values should be specified. Their proposal leverages mutation analysis to determine how often each variable is able to reveal a mutant

and the variables and outputs are ranked in terms of fault finding. The approach is evaluated with four systems and shows improvements from 5% to 30% with a best case of 145.8% compared to only using the original output as an oracle and random data set selection.

Most of these works leverage the observation during test executions to create regression oracles and new assertions. These principles can be also used to study extreme transformations. A difference between the execution of the original program and a variant of the program with an undetected extreme transformation, can lead to a potential flaw in the oracle.

1.2.5 Reachability-Infection-Propagation

The *Reachability, Infection, Propagation* model (RIP) [48, 49, 24] states the necessary conditions to detect a fault. That is: (i) the fault should be reached/executed (ii) the program state should be changed/infected after the fault is executed (iii) the program infection should propagate to the output.

This model provides a framework that can be leveraged to analyze faults in their context and produce test suite improvements.

Voas introduced the *propagation*, *infection* and *execution* (*PIE*) analysis [81]. It operates in three phases: estimate the frequency at which random inputs execute different code locations; mutate these locations to produce a different program state from the previously observed values; estimate the probability that altered program states change the output of the program. Subsequently, Voas and Miller [83, 82] discussed the notion of *testability* in relation to the PIE analysis: "*the likelihood that the code can fail if something in the code is incorrect*". The authors express concerns about the loss of observation ability linked to information hiding in object-oriented programs.

Li and Offutt [45] evolve the RIP model into *Reachability Infection Propagation Revealability*, (*RIPR*) to include the notion that the oracle in automated tests must *reveal* failures. They explore RIPR for model-based testing and devise several *test oracle strategies*, i.e., rules to specify which program states to check.

Lin *et al* [46] recently proposed D-RIP, a combination of RIP analysis with domain-based testing. The authors estimate the degree at which mutants can be detected by different test selection strategies considering different test assessment criteria. Their goal is to identify stubborn mutants and rank mutants by difficulty of detection. These authors conclude that the no propagation of the program infection is the main reason why a mutant is not detected.

The work of Androutsopoulos et al. [4] focuses on how faulty program states propagate to observable points in the program. They show that one in ten test inputs fails to propagate to observable points. The authors provide an information theoretic formulation of the phenomenon through five metrics and experiment with 30 programs and more than 7M test cases. They state that better understanding the causes of failed error propagation leads to better testing.

1.2.6 Tools and Industrial exploitation of mutation testing

Practitioners and researches have created several tools implementing mutation testing. Papadakis and colleagues [56] report the existence of 76 tools introduced after 2001.

The paper reports 17 tools for Java. However, most of these tools were created for specific academic purposes or they are discontinued and not available anymore. Most of them do not integrate with exiting build systems or can not be extended beyond their basic functionalities. Major [38], LittleDarwin [57] and PITest [13] stand apart from the rest in those terms. Major and PITest allow extending their functionalities with new set of mutation operators. LittleDarwin and PITest integrate with Ant, Gradle and Maven. PITest targets compiled bytecode which allows to use the tool in other JVM languages. Of these three alternatives PITest is the most mature, popular and extensible.

PITest [13] was created by a developer, to assist developers in the creation of unit test cases. This tool implements most traditional mutation operators and performs the transformations at the bytecode level. The tool executes only the test cases covering each mutant. It also sorts each test cases so that the ones requiring less time are executed first. PITest is able to executed the tests in parallel. The tool can be extended with plugins to change how test cases are discovered and prioritized and even to provide an alternative implementation of the mutation operators.

Delahaye and Bousquet [18], then Kintis et al. [41] compare mutation tools from the usability point of view concluding that PITest is one of the best alternatives for concurrent execution and adaptability to distinct requirements. In a follow-up paper, Laurent and colleagues [44] propose to improve PITest with an extended set of mutation operators shown to obtain better results.

More recently, Gopinath et al. [30] performed a comprehensive analysis of three software tools, including PITest, with 27 projects. They run several statistical analyses to compare the performance of the tools considering projects and tool characteristics against raw mutation score, a refined mutation score to mitigate the impact of equivalent mutants and the relationship among mutants. They conclude that PITest is slightly better than the other tools. They also state that the specificity of each project have a high impact on the effectiveness of the mutation analysis.

Despite the existence of several mutation testing tools there is no wide adoption of the technique by industry practitioners. One of the few large scale industrial experiences is reported by Petrovic and Ivankovic at Google [59].

These authors explain that the Google monolithic repository contains about two billion lines of code and on average, 40000 changes are commited every workday. 60% of those changes are created by automated systems. In this environment it is not practical to compute a mutation score for the entire codebase and it is very hard to provide actionable feedback from this analysis.

Most changes in the repository pass through a code review process. Hence, the authors argue that this is the best location in the workflow to provide feedback about the mutation analysis and eliminate the need for developers to run a separated program and act upon its output. So, live mutants are shown as *code findings* in code reviews.

To make the mutation analysis feasible the proposed system creates at most one mutant by covered line. The mutation operator is selected at random from a set of available options. To further reduce the number of mutants, they classify each node of the Abstract Syntax Tree (AST) as important or non-important (*arid*). To do this, they maintain a curated collection of simple AST nodes classified by experts, that keeps updating with the feedback of the reviewing process. Compound nodes are classified as arid if all their children are arid. Uninteresting nodes may be related to logging, non-functional properties and nodes seen as “axiomatic” for the language and thus the mutants are trivially killed. This selection may suppress relevant live mutants but the authors state that the tradeoff between correctness and usability of the system is good. The number of potential mutants is always much larger than what can be presented to reviewers.

The system analyses programs written in C++, Java, Python, Go, JavaScript, TypeScript and Common Lisp. It has been validated with more than 1M mutants in more than 70K diffs. 150K live mutants were presented and 11K received feedback from developers. 75% of the findings with feedback were reported to be useful. The authors also observed interesting differences related to the survival ratio of mutants when contrasted with the programming language and mutation operator.

In a follow-up paper [60], these authors explain that there are killable mutants that become unproductive, in a sense that they could be killed with a test case that developers deem unnecessary and represent a waste of time. These could be, for example, mutants that break the stablished coding practices of the company or changing $>$ by $>=$ in float point comparisons, among others. These authors also corroborate the observations made by Coles [12] that equivalent mutants can point to redundancy on the application code that could be removed via refactoring. They also argue that mutants should be presented to developers at the commit level, which is the “standard unit of work for a developer”. Finally they argue that the focus should not be to produce a mutation adequate test suite, that is, to reach a high mutation score. The goal should rather be to help make the test suite better. In their own words “this returns to the roots of mutation testing: providing hints for the practitioner programmer”.

1.2.7 Helping testers

Some works in the specialized literature provide developers with concrete hints to improve their tests. The following set of papers find the conditions inputs should meet to execute a change in the code and propagate the effects of the change to observable points. The goal of the authors is to help developers in the creation of new test cases exercising those changes.

Apiwattanapong *et al* [5] target the problem of finding test conditions that could propagate the effects of a change in a program to a certain execution point. Their method takes as input two versions of the same program. First, an alignment of the statements in both versions is performed.

Then, starting from the originally changed statement and its counterpart in the new version, all statements whose execution is affected by the change are gathered up to a certain distance. The distance is computed over the control and data dependency graph. A partial symbolic execution is performed over the affected instructions to retrieve the states of both program versions, which are in turn used to compute testing requirements that can propagate the effects of the original change to the given distance. As said before, the method does not deal with test case creation, it only finds new testing conditions that could be used in a separate generation process, manual or automatic, and is not able to handle changes to several statements unless the changed statements are unrelated. The approach is evaluated on Java translations of two small C programs (102 Loc and 268 LoC) originally included in the Siemens program dataset [34]. The authors conclude that, although limited to one change at a time, the technique can be leveraged to generate new test cases during regular development.

Santelices *et al* [65] continue and extend the previous work by addressing changes to multiple statements and considering the effects they could have on each other. In order to achieve this they do not compute state requirements for changes affected by others. This time, the evaluation is done in one of the study subjects form their previous study and two versions of *Nanoxml* from SIR.

In another paper [66] the same authors address the efficiency problems of applying symbolic execution. They state that limiting the analysis of affected statements up to a certain distance from changes reduces the computational cost, but scalability issues still exist. They also explain that their previous approach often produces test conditions which are unfeasible or difficult to satisfy within a reasonable resource budget. To overcome this, they perform a dynamic inspection of the program during test case execution over statically computed slices around changes. The technique is evaluated over five small Java programs, comprising *Nanoxml* with 3 KLoC and translations of C programs from SIR having between 283 LoC and 478 LoC. This approach also considers multiple program changes. Removing the need of symbolic execution leads to a less expensive method. The authors claim that propagation-based testing strategies are superior to coverage-based in the presence of evolving software.

Song and colleagues [71] have developed UnitPlus, a tool to assist developers in test creation. The tool recommends observer methods that could be used to verify the effects of methods that change the state of the receiver. The already mentioned ReAssert [16] presents its output as suggestions for developers instead of automatically replacing the existing test cases.

Delplanque *et al* [20] propose DrTest, a tool to find *green rotten tests* in Pharo programs by combining static and dynamic test code analysis. *Green rotten tests* are those whose assertions are not executed and therefore do not fail and do not properly verify the program under test. These authors theorize that not executing the assertion could be a cause of pseudo-tested methods.

1.3 Conclusion

In this chapter we settled the background for the rest of the document. We reviewed how mutation testing is used to assess test suites and automatically improve test cases. We have discussed extreme mutations and the results from Niedermayr and colleages. We have also described the RIP model which organizes the analysis of software faults. Our main goal is to provide developers with concrete, targeted an easy to understand feedback to improve their test suites. We leverage extreme transformations, pseudo-tested methods and the RIP analysis to achieve this goal. In Chapter 2 we present, Descartes, our own tool to detect pseudo-tested methods and compare both approaches mutation testing and extreme transformations. In Chapter 3 we expand the study on pseudo-tested methods and check their utility in practice. In Chapter 4 we propose a novel analysis inspired by the RIP model to help developers understand the results produced by Descartes and how to fix testing issues revealed by undetected extreme transformations.

CHAPTER 2

DESCARTES A PITEST PLUGIN FOR EXTREME TRANSFORMATIONS

Software development escapes from the solitude of a single programmer in a closed environment. Software is built from the interactions of teams composed of dozens of developers. These developers are constantly refactoring the code, adding new features and committing their changes to distributed version control systems several times a day. These code repositories are automatically monitored by Continuous Integration (CI) servers that verify the quality of the code. Developers expect a concise and actionable feedback from the CI that helps them to improve their code. In most cases, developers also create their own unit tests. Since test suites are also software artifacts, the CI can analyze and provide feedback about their quality.

In this thesis we investigate extreme transformations and pseudo-tested methods in order to point to concrete testing issues and help developers to improve their test suites. This novel line of research mixes static and dynamic program analyses with the generation of concrete feedback to developers. It is, by essence, a research line based on empirical methods. The relevance of this approach can be assessed only through sound experiments with real-world software applications under continuous development and with their developers. Consequently, we need a tool that can be inserted in such a development environment. To achieve this goal we have developed Descartes, a tool built on top of PITest, able to study extreme transformations and detect pseudo-tested methods in Java projects.

This chapter addresses the first two objectives of this thesis: create a robust tool for extreme transformations and empirically compare extreme transformations with traditional mutation testing. Here we introduce Descartes, the main ideas behind its development and how it is used. In Section 2.1 we briefly describe the tool, how it fits in the PITest framework and the transformations that the tool implements. In Section 2.2 we perform an empirical comparison, between extreme transformations and traditional mutation testing using Descartes and Gregor, PITest's default mutation engine. Finally, in Section 2.3.1 we present an industrial experience of the exploitation of Descartes by XWiki, a software development company.

2.1 An overview of Descartes

Descartes is a tool that automatically detects pseudo-tested methods with the help of extreme transformations. It has been directly inspired by the work of Niedermayr and colleagues [51]. The requirements for Descartes demanded the analysis of Java projects using build systems such as Maven and Gradle. It is also required that the tool could be integrated in modern development workflows *i.e.* CI servers. Therefore, the natural choice was to base Descartes on PITest to leverage its maturity and functionalities.

Descartes heavily exploits PITest's extensible architecture. By using and extending PITest we obtain a solid and multithread infrastructure for unit test discovery and execution, support for Ant and the already mentioned Gradle and Maven build systems, support for JUnit and TestNG test libraries and the support of an engaged community of researchers and developers contributors all committed to the quality of the tool. Relying on PITest also allowed a faster adoption of Descartes in production.

The main challenges developing Descartes are: (i) The lack of documentation describing how to create extensions for PITest, the code of the tools and plugin examples [1] provided by the author are the main source of information; (ii) the design and implementation of meaningful abstractions for extreme transformations and their interaction with the rest of the PITest framework; (iii) maintaining the tool up to date with the regular evolution and releases of PITest; (iv) making the tool

[1] https://github.com/hcoles/pitest-plugins

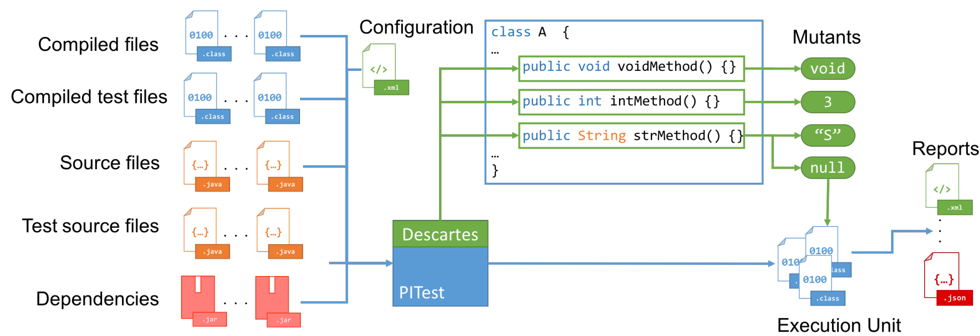


Figure 2.1: Interconnection between PITest and Descartes.

useful for developers by reducing the number of uninteresting findings and providing a meaningful output.

Descartes is in fact a collection of extensions for PITest. Its main component is a mutation engine, that is, an extension that provides an alternative set of mutation operators, in this case, extreme transformations. It also includes an extension to avoid creating transformations in methods that are not usually targeted by developers in test cases as they are trivial or non-important and custom reporting extensions. In the following sections we provide a brief description of these components.

2.1.1 Engine for extreme transformations

The main component of Descartes is an alternative mutation engine for PITest. In PITest's jargon, a mutation engine is a plugin that handles the discovery and creation of mutants. Such a plugin should also manage a set of mutation operators, which are models of the transformations to be performed. The mutation operators provided by Descartes actually implement extreme transformations as described in

Figure 2.1 illustrates the interaction between PITest and our mutation engine. PITest handles the inspection of the target project to discover all dependencies, creates execution units composed by the mutants and the tests to be executed, and ultimately runs the test cases. The mutation engine leverages all these functionalities and is in charge of discovering the places in code where the transformations/mutations can be performed and creates the program variants/mutants.

The mutation engine can be configured to use a custom set of extreme transformations. The engine is able to transform any Java method, except constructors. As of Descartes 1.2.6 [2], the tool includes the transformation of void methods, methods returning reference objects by replacing the code with **return null** and methods returning arrays by replacing the result value with an empty array. The engine allows to specify the constant values to use for methods returning values of primitive types or `String`. It also includes two special operators, one to handle methods returning instances of `Optional` and methods returning instances of classes having a constructor without parameters. The full list of transformations is shown in Table 2.1.

To the best of our knowledge, Descartes is the only available alternative to the default mutation engine provided by PITest. Our project could be used as an additional supporting material for those who are willing to create their own extensions.

2.1.2 Stop-methods

Petrovic and colleagues [60] discuss that unproductive findings reported by a mutation testing tool represent a waste of time for developers. Descartes includes an extension that discards transformations which are arguably not interesting for developers.

The extension filters out all transformations that would be created in methods that developers usually consider as trivial or methods generated by the compiler. It also excludes methods whose transformation could result in an equivalent mutant. These are what we call *stop methods*. The full list of stop methods is shown in Table 2.2.

[2] https://github.com/STAMP-project/pitest-descartes/releases/tag/descartes-1.2.5

Table 2.1: Extreme transformations included in Descartes

Operator	Description	Configuration	Transformation
`null`	Replaces the instructions by **return null**	`null`	`Object m(){ return null; }`
`void`	Removes all instructions in void methods	`void`	`void m(){ }`
`empty`	Makes the method return an empty array	`empty`	`int[] m(){ return new int[0]; }`
Constant values	Makes the method return an specified constant value of the appropriate type	`true, 1, 2L, 3.0f, 4.0, 'a', "literal`	`int m(){ return 1; }`
`new`	Uses a constructor without parameter	`new`	`List m(){ return new ArrayList(); }`
`optional`	Returns an empty optional	`optional`	`Optional<Integer> m() { return new Optional<>.empty(); }`

Table 2.2: Stop-methods

Description	Example
Empty void methods	`public void m(){}`
Methods generated to support enum types (`values` and `valueOf`)	
`toString` methods	
`hashCode` methods	
Methods annotated with @Deprecated or belonging to a class with the same annotation	`@Deprecated public void m(){...}`
Synthetic methods *i.e.* generated by the compiler for internal purposes	
Simple getters	`public int getAge(){ return this.age; }`
Simple setters including also fluent simple setters	`public void setX(int x) { this.x = x; }` `public A setX(int x) { this.x = x; return this; }`
Methods returning a literal value	`public double getPI(){ return 3.14; }`
Methods implementing simple delegation	`public int sum(int[] a, int i, int j) { return this.adder(a, i, j); }`
Static class initializers	
Methods that only return **this**	`public A m(){ return this; }`
Methods that only return the value of a real parameter	`public int m(int x, int y){ return y; }`
Setters generated for data classes in Kotlin	

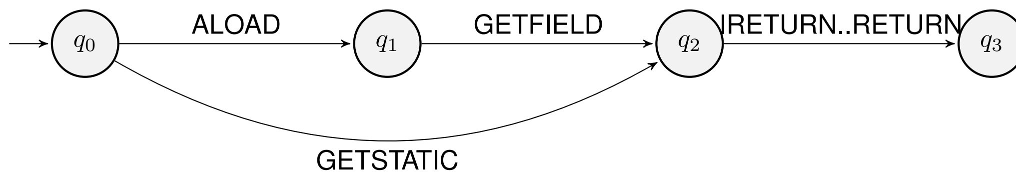


Figure 2.2: Finite automaton to detect simple getter methods

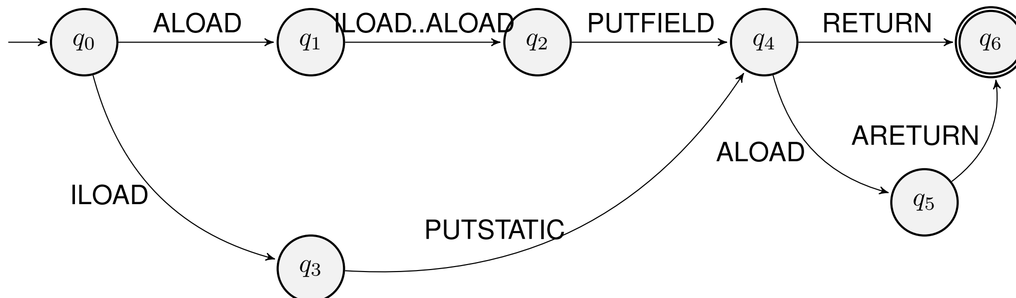


Figure 2.3: Finite automaton to detect simple setter methods

`toString` methods, `hasCode` methods, most compiler generated methods, deprecated methods and class initializers can be detected by inspecting the metadata of each method. These methods have a particular signature, or their names include special characters or they have been marked with a special annotation such as `@Deprecated`.

All other stop-methods have to be detected by inspecting their code. Our extension achieves this goal by checking the code of each method against a set of Deterministic Finite Automata. Each automaton encodes the structure of a category of stop-methods and takes as input a string whose symbols are JVM opcodes. Figures 2.2 and 2.3 show the transition function of the automata that detect simple getters and setters respectively.

2.1.3 Output of Descartes

Descartes includes three additional extensions to produce reports with the output of the analysis. One extension generates the same output as the default reports from PITest, but using the `JSON` format. The other two reporting extensions are dedicated to pseudo-tested methods.

Niedermayr *et al* define pseudo-tested methods as those for which no extreme transformation is detected by the test suite. The two reporting extensions monitor the test executions and group all the transformations by the method in which they are created. Then the extensions classify each method specifying whether they are pseudo-tested or not.

One of the extensions produces a `JSON` file, meant to support the incorporation of the tool in larger workflows, in the CI, for example. This file contains all the static information of the method: name, signature, declaring class, code location. It also includes all the test cases found to cover the method and whether the method is pseudo-tested. The report contains the information relative to the extreme transformations: how many of them were created for the method, if they were detected or not and the operator, as shown in Table 2.1 that were used to create the program variant.

The other extension produces the same information but in a human readable presentation. Figure 2.4 shows an example of a report produced by Descartes for a method in the Apache Commons CLI project [3].

These two extensions report the pseudo-tested methods in the project but they also include all undetected extreme transformations. Methods with mixed results, that is where some extreme transformations were detected and at the same time others were not, may also indicate the presence of testing issues. This will be explored in Chapter 4.

[3] https://github.com/apache/commons-cli

org.apache.commons.cli.AmbiguousOptionException

createMessage(java.lang.String, java.util.Collection)

This method is **pseudo-tested**.

The body of this method has been replaced by:

- `return "A";`
- `return "";`
- `return null;`

Yet, no test case detected any of those changes.

The following test cases execute the method:

- org.apache.commons.cli.PosixParserTest.testAmbiguousPartialLongOption1(org.apache.commons.cli.PosixParserTest)
- org.apache.commons.cli.DefaultParserTest.testAmbiguousPartialLongOption3(org.apache.commons.cli.DefaultParserTest)
- org.apache.commons.cli.DefaultParserTest.testAmbiguousPartialLongOption2(org.apache.commons.cli.DefaultParserTest)
- org.apache.commons.cli.DefaultParserTest.testAmbiguousPartialLongOption4(org.apache.commons.cli.DefaultParserTest)
- org.apache.commons.cli.DefaultParserTest.testAmbiguousPartialLongOption1(org.apache.commons.cli.DefaultParserTest)
- org.apache.commons.cli.bug.BugCLI252Test.testAmbiguousOptionName(org.apache.commons.cli.bug.BugCLI252Test)
- org.apache.commons.cli.PosixParserTest.testAmbiguousPartialLongOption3(org.apache.commons.cli.PosixParserTest)
- org.apache.commons.cli.PosixParserTest.testAmbiguousPartialLongOption2(org.apache.commons.cli.PosixParserTest)

Figure 2.4: An example of a human readable report produced by Descartes

2.2 Descartes VS Gregor

Niedermayr *et al* [51] affirm that extreme transformations generate much less program variants than traditional mutation testing. In this section we quantify this fact for 21 Java open-source projects. We compare the execution of Descartes with Gregor, the default mutation engine for PITest. Gregor implements most traditional mutation operators [4]. These operators work at the instruction level similar to the example in Listing 1.4.

The 21 selected projects are shown in Table 2.4. This selection is a combination of the projects studied in the original paper of Niedermayr and colleagues, projects studied in the related literature, projects from our industrial partners and projects with a mature development history. All these projects use Maven as main build system, JUnit as main testing framework and are available form a version control hosting service, mostly Github. We will provide more insight about this selection in Chapter 3. We provide the code revision used for each project in Appendix A.

We execute both mutation engines in all projects. For each observation we observe the time it takes to complete the analysis, how many program variants/mutants both engine create, how many of them are actually executed by the test suite and how many of them are detected/killed by the existing test cases. We used in the analysis all traditional mutation operators available for Gregor. With Descartes we used the same mutation operators as Niedermayrs et. al. [51] plus two additional transformations, one to return **null** for reference types and another to return an empty array. The full list of extreme mutation operators is shown in table 2.3.

Table 2.3: Extreme mutation operators used in the comparison.

Method type	Transformations
void	Empties the method
Reference types	Returns null
boolean	Returns true or false
byte,short,int,long	Returns 0 or 1
float,double	Returns 0.0 or 0.1
char	Returns ' ' or 'A'
String	Returns "" or "A"
T[]	Returns new T[]{}

Table 2.4 shows the metrics we recorded for the comparison. For each mutation engine the table shows the execution time and number of mutants/program variants created. The "*Covered*" columns show how many of them are actually executed by the test suite and planted in methods that were transformed by both engines. This distinction removes from the comparison mutants that Gregor may create in methods not analyzed by Descartes, and vice-versa. For example,

[4] The full list is available here: http://pitest.org/quickstart/mutators/

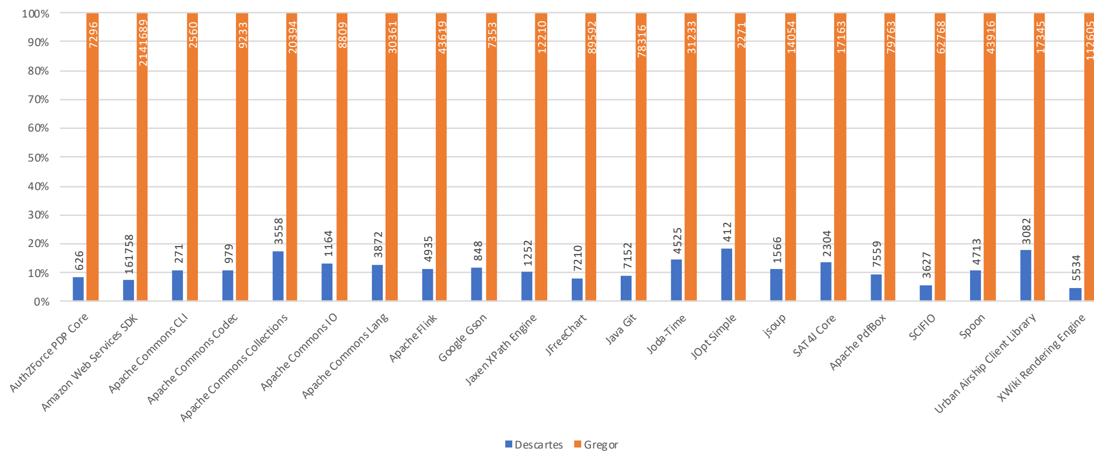


Figure 2.5: Visual comparison between the number of mutants/program variants create by both engines

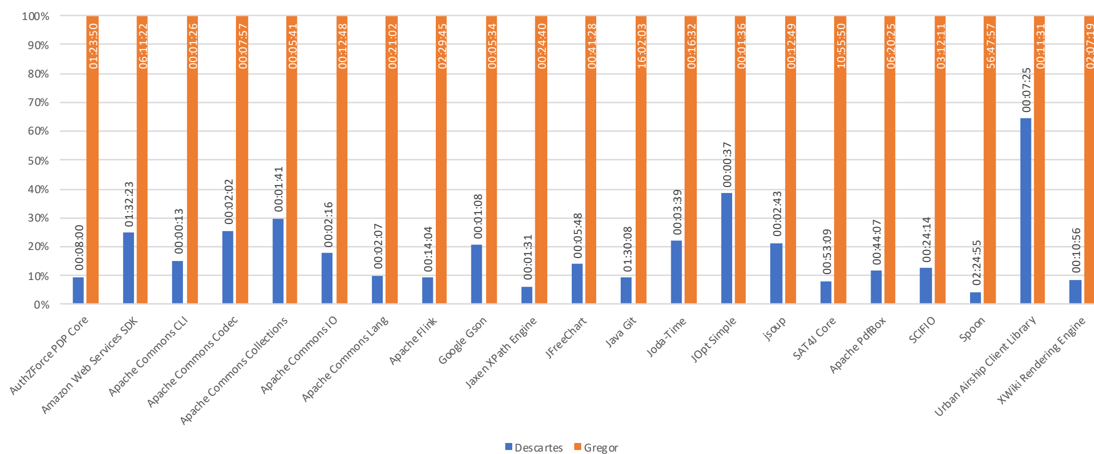


Figure 2.6: Visual comparison between the execution time of both engines

mutants created in constructors and other stop-methods are left out. The "*Killed*" columns contain the number of mutants from the respective "*Covered*" column that were detected/killed by the test suite. The "*Score*" columns show the corresponding mutation score, that is the ratio of "*Killed*" to "*Covered*".

For a better visual perspective Figure 2.5 and Figure 2.6 compare the number of mutants and execution time of both engines in the entire set of projects. In both figures we show the relative reduction achieved by Descartes wit respect to Gregor. The values measured for Gregor has been normalized to 100% so the values of Descartes are shown in proportion.

One can observe that Descartes creates much less mutants than Gregor which is reflected in the difference between the times to execute the analysis of each engine. In all cases, Descartes completed the task in much less time. Some interesting contrasts in this matter come from projects like `Spoon` where Descartes took a little less than two hours and a half while Gregor took more than 56 hours, `Java Git` with one hour and a half for extreme mutation and 16 hours for Gregor and `Jaxen XPath Engine` with less than two minutes against nearly 25 minutes. While the number of mutants created and covered affects the execution time, the tests themselves play an important role as they can involve heavy computation. Take, for example, the difference between `Apache Commons Lang` and `SCIFIO` with similar numbers of mutants and very different execution times. This comparison in terms of execution time is just an illustration of the actual reduction. There are expected fluctuations in time from one execution of PITest or Descartes to another. For an accurate comparison one should run the tools several times per project and provide statistical evidence.

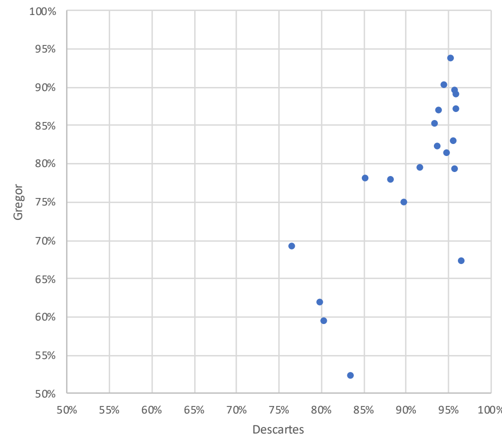


Figure 2.7: Visual correlation between the scores of both engines

```java
public static boolean isValidXmlChar(int ch) {
    return (ch == 0x9)
    || (ch == 0xA)
    || (ch == 0xD)
    || (ch >= 0x20 && ch <= 0xD7FF)
    || (ch >= 0xE000 && ch <= 0xFFFD)
    || (ch >= 0x10000 && ch <= 0x10FFFF);
}
```

Listing 2.1: Real example of a pseudo-tested method.

As for the scores, one can notice that there is a certain correlation between the values obtained by both engines. Figure 2.7 shows a scatter plot, in which each point represents a project. The coordinates for each point are given by the scores, the x axis corresponds to the score from Descartes while the y represents the score from Gregor. The figure corroborates the tendency for a positive monotonic correlation between both scores, which means that, if the score with Gregor is high, it is more likely that the score with Descartes will be also high. The Spearman correlation coefficient results in 0.6 for the projects studied with a *p-value* of 0.003, which indeed indicates that there is a moderate positive correlation. Anyways, there are cases such as `SCIFIO` and `XWiki Rendering Engine` which produce a medium to low mutation score with Gregor and scores above 83% with Descartes. We can conclude that, in general, well tested projects, taking the traditional mutation score as the main assessment, will also detect or kill most extreme transformations. Badly tested projects, or projects in their initial state of development could benefit the most from Descartes.

2.2.1 Pseudo-tested methods

The results of Descartes are not limited to produce a score for a given project. The proposal of Niedermayr *et al* [51] classifies methods according to the extreme transformation detection. A method is said to be **pseudo-tested** if it is covered by the test suite but no related extreme transformation is detected. These methods are the worst tested in the code base. The detection of extreme transformations provides a framework to find such methods more efficiently than traditional mutation testing.

Listing 2.1 shows a method belonging to one of the projects included in table 2.4. It was found to be pseudo-tested by Descartes. Only two extreme mutations are required to detect that the value of this method is not correctly verified by the test suite, if verified at all, while Gregor created 45 mutants. This is an example of the utility of extreme transformations.

Among the pseudo-tested methods discovered by Descartes we found cases of apparent relevance. Code listing 2.2 shows a pseudo-tested method found in `Apache Commons Collections` declared in a hash map implementation.

All the instructions of this method are covered across 38 test cases and the method itself is executed 11593 times while running the test suite. The method enlarges the internal structure of the hash map to match a new capacity. It is called when the number of collisions in the map reaches a defined threshold. It is clearly a relevant method for the functionalities of the class. Despite its importance, it is pseudo-tested. There are some factors that affect the testability of this method. First, it is declared in an abstract class. Second, it is a private method. It can not, and should not, be directly verified by any test case. However, its effects play an important role in the class, so they should be assessed. This method affects the state of the receiver, namely the values of the `threshold` and `data` fields, both protected and therefore accessible to the test cases. Yet, no assertion checks those modifications.

```java

abstract class AbstractHashedMap<K, V> {
  ...
  protected void ensureCapacity(final int newCapacity) {
    final int oldCapacity = data.length;
    if (newCapacity <= oldCapacity) {
        return;
    }
    if (size == 0) {
      threshold = calculateThreshold(newCapacity, loadFactor);
      data = new HashEntry[newCapacity];
    }
    else {
      final HashEntry<K, V> oldEntries[] = data;
      final HashEntry<K, V> newEntries[] = new HashEntry[newCapacity];
      //Rehash code
      ...
      threshold = calculateThreshold(newCapacity, loadFactor);
      data = newEntries;
    }
  }
  ...
}
```

Listing 2.2: pseudo-tested method in `Apache Commons Collections`

Nevertheless, the result of Descartes is coarse-grained. Methods where extreme transformations are detected are not exempt from having testing issues. Listing 2.3 shows a real example of a method where all extreme transformations were detected but Gregor created mutants that survived the analysis. In particular one of the traditional mutation operators changed the value of `Long.MIN_VALUE` in line 2. The modification was unnoticed by the test suite, which indicates that the corner case is not being tested. This level of detail can not be reached with the use of extreme mutation alone.

```java
public long subtract(long instant, long value) {
    if (value == Long.MIN_VALUE)
        throw new ArithmeticException(...);
    return add(instant, -value);
}
```

Listing 2.3: Example of a non pseudo-tested method.

In Chapter 3 we expand the discussion about pseudo-tested methods. We perform a conceptual replication of the work of Niedermayr and colleagues and analyze whether these methods are valid hints to improve existing test cases. We also provide a set of testing issues found with the help of Descartes in real and well tested open-source projects.

Table 2.4: List of projects used to compare both engines, the execution time for the analysis, the number of mutants created, mutants covered and placed in methods targeted by both tools, mutants killed and the mutation score.

	Descartes					Gregor				
Project	Time	Created	Covered	Killed	Score	Time	Created	Covered	Killed	Score
AuthZForce PDP Core	0:08:00	626	378	358	94.71	1:23:50	7296	3536	3188	90.16
Amazon Web Services SDK	1:32:23	161758	3090	2732	88.41	6:11:22	2141689	17406	13536	77.77
Apache Commons CLI	0:00:13	271	256	246	96.09	0:01:26	2560	2455	2183	88.92
Apache Commons Codec	0:02:02	979	912	875	95.94	0:07:57	9233	8687	7765	89.39
Apache Commons Collections	0:01:41	3558	1556	1463	94.02	0:05:41	20394	8144	7073	86.85
Apache Commons IO	0:02:16	1164	1035	968	93.53	0:12:48	8809	7633	6500	85.16
Apache Commons Lang	0:02:07	3872	3261	3135	96.14	0:21:02	30361	25431	22120	86.98
Apache Flink	0:14:04	4935	2781	2373	85.33	2:29:45	43619	21350	16647	77.97
Google Gson	0:01:08	848	657	617	93.91	0:05:34	7353	6179	5079	82.20
Jaxen XPath Engine	0:01:31	1252	953	921	96.64	0:24:40	12210	9002	6041	67.11
JFreeChart	0:05:48	7210	4686	3775	80.56	0:41:28	89592	47305	28080	59.36
Java Git	1:30:08	7152	5007	4507	90.01	16:02:03	78316	54441	40756	74.86
Joda-Time	0:03:39	4525	3996	3827	95.77	0:16:32	31233	26443	21911	82.86
JOpt Simple	0:00:37	412	397	379	95.47	0:01:36	2271	2136	2000	93.63
jsoup	0:02:43	1566	1248	1197	95.91	0:12:49	14054	11092	8771	79.08
SAT4J Core	0:53:09	2304	804	617	76.74	10:55:50	17163	7945	5489	69.09
Apache PdfBox	0:44:07	7559	3185	2548	80.00	6:20:25	79763	32753	20226	61.75
SCIFIO	0:24:14	3627	1235	1158	93.77	3:12:11	62768	19615	9496	48.41
Spoon	2:24:55	4713	3452	3171	91.86	56:47:57	43916	34694	27519	79.32
Urban Airship Client Library	0:07:25	3082	2362	2242	94.92	0:11:31	17345	11015	8956	81.31
XWiki Rendering Engine	0:10:56	5534	3099	2594	83.70	2:07:19	112605	50472	26292	52.09

2.3 Development and adoption

At the moment of writing this document the last stable version of Descartes is 1.2.6. The tool has been released to Maven Central eight times starting from its version 1.1. In the lapse of three years 99 issues have been opened of which 67 have been closed. Other collaborators have created 11 pull request 10 of them have been merged into the code base. The project has been forked 13 times.

Descartes is an active component of her two sister projects DSpot and Reneri. DSpot [14] is a tool that automatically improves/amplifies existing test cases. It uses the ratio of undetected extreme transformations as a fitness function for the improvement process. Reneri analyzes the output of Descartes and produces test improvement suggestions for developers. It uses Descartes to replay the previously undetected transformations. We will present Reneri in Chapter 4. In the same chapter we will evaluate the effectiveness of DSpot against undetected extreme transformations.

Niedermayr and Wagner [52] report the use of the tool in their experiments to explore the correlation between the stack distance and test case effectiveness.

By leveraging PITest, Descartes integrates with Ant, Maven and Gradle. Apart from this, our industrial partners have developed other integration alternatives targeting the CI. The OW2 consortium developed a Gitlab plugin that creates issues in the tracker based on the output of Descartes. We have also developed a Github application, that makes use of the Github checks API. This application triggers a remote Descartes execution after a pull request and annotates the output of the Github checks with the list of pseudo-tested methods. Engineering, a software service company, developed a Jenkins plugin for Descartes. This plugin keeps track of the ratio of undetected extreme mutations and the number of pseudo-tested methods in a project throughout the development history.

The collaboration with our industrial partners has been a key element in the development of Descartes. Companies as TellU, Atos, ActiveEon and XWiki SA. have tried the tool in their codebase and have provided valuable and key feedback on issues and what kind of output and transformations are most relevant for developers.

Most of these partners have limited themselves to manually launching the tool and analyzing the output. However, XWiki proposed an original methodology which incorporates the tool in their CI and ensures the improvement of their test suite in time.

2.3.1 XWiki: an industry case study with Descartes

XWiki[5] is an open-source Enterprise Knowledge Management and Collaboration Solution. XWiki is developed by an open-source community where the company XWiki SAS is the main contributor. The main codebase of XWiki includes three multi-module Maven projects: `xwiki-commons`[6], `xwiki-rendering`[7] and `xwiki-platform`[8]. These three projects are mainly written in Java and together contain more than 340K lines of application code (excluding comments) and around 170K of lines of test code distributed in 196 Maven modules. All these test cases achieve a 74% statement coverage ranging from 55% to 95% on each module. From October 13th, 2006, the initial commit, to September 17th 2019 the three aforementioned projects accumulated 45114 commits. The contributors made 11 commits per day on average. So, these projects are in constant development.

Software quality at XWiki

The XWiki solution is a complex and extensible platform. Its developers pay special attention to code and test quality. They have set a highly sophisticated infrastructure which includes the integration with tools like Jira as an issue tracker, Sonarqube for code analysis, Nexus as an artifact repository and Jenkins, as a continuous integration server. The XWiki community has put in place strict practices for unit test development. For example, each test class should test only a single class in isolation. The test cases must not interact with the environment, that is databases,

[5] https://xwiki.org
[6] https://github.com/xwiki/xwiki-commons
[7] https://github.com/xwiki/xwiki-rendering
[8] https://github.com/xwiki/xwiki-platform

file systems and alike. Test cases must not require any configuration to run nor output anything to the standard error and standard output. Every commit on any of the XWiki projects triggers the execution of the test suite in the CI.

To promote the continuous improvement of their very well crafted test cases, the XWiki community has devised the following approach. A job in the CI monitors the code coverage when the test suite is executed. Each Maven module in the codebase has a predefined threshold and the code coverage can not decrease below this value, otherwise the build fails. In this way, if a developer adds some code she has to also provide new tests cases so the coverage ratio remains above or equal the predefined value. If a developer achieves a coverage above threshold, then she is given the possibility to raise the threshold value for the module. In this way it is ensured that the code coverage never decreases, this is what they call the *Ratchet Effect*. This strategy has led to an effective use of the code coverage metric. They report an increase of around 6% of the global code coverage in little less than 11 months[9].

Descartes at XWiki

The XWiki projects started using Descartes after September 2017. At first, the tool was used manually to target specific modules and find testing issues. During this time, the contributors created 20 commits fixing or adding new test cases following the results produced by Descartes.

In March, 2018 the team proposed to take Descartes to the CI. They implemented a strategy similar to the one already in practice for code coverage. Each module is configured with a threshold for the ratio of undetected extreme transformations. If the score produced by Descartes is below the threshold then the build fails. Unlike code coverage, this CI job is not triggered after every commit as it takes several hours to complete the execution.

After implementing this CI approach, XWiki developers have produced 21 additional commits fixing testing issues found by Descartes. In 12 occasions the CI job has failed due to a drop in the score and prevented the test quality to decrease.

Developers made 41 commits in total. 20 before the CI integration and 21 after the inclusion of Descartes in their build workflow. These commits increased by 2% in average the code coverage of the module in which they were created. In total , XWiki contributors added 66 new test classes, modified 22 classes and directly edited more than 700 assertions in the code.

One of the main issues in the adoption of Descartes has been the realization that the score does not evolve in the same way as code coverage does. Listing 2.4 illustrates this phenomenon.

Method `m` (line 2) has been refactored into a new version (line 9) where the validation is performed by method `check` (line 14). Let us assume that `m` has only been tested with a value of `x` greater than zero. In the original version of `m` 2/3 lines are covered. In the refactored code 3/4 lines are covered.

However, the introduction of `check` also induces the creation of a new extreme transformation that removes the code of this method. This transformation will pass undetected as the case in which `x` is lower than zero is never tested.

So, in this example, developers improved their code, the code coverage improves but the score computed by Descartes decreases. In the case of XWiki this means that the build in their CI fails.

To our view this is a positive result since a testing issue that passed unnoticed before was detected after the refactoring. But it requires a change of mindset for developers.

```
1  // Before refactoring
2  void m (int x) {
3    if ( x < 0 )
4      throw new Exception();
5    doSomething();
6  }
7
8  // After refactoring
9  void m(int x) {
10   check();
11   doSomething();
12 }
13
```

[9]https://massol.myxwiki.org/xwiki/bin/view/Blog/ComparingCloverReports

```
14 void check(int x) {
15   if (x < 0)
16     throw new Exception();
17 }
```

Listing 2.4: A code refactoring action that decreases the ratio of undetected extreme transformations

2.4 Conclusion

In this chapter we presented Descartes, a tool to create extreme mutations and discover pseudo-tested methods in Java projects. By leveraging the functionalities of PITest, Descartes could make a short transit from laboratory prototype to a production ready tool. We illustrated in 21 open-source projects how the use of extreme transformations compare to traditional mutation testing in terms of number of mutants and execution time. We also showed that the score computed by both approaches is positively correlated. While the improvement and score depends on the specificities of each project, the reduction using extreme transformations in noticeable and can make the analysis more scalable. We also presented how Descartes is being used by an industrial actor as part of their CI infrastructure. We also illustrated the kind of testing issues that pseudo-tested methods can reveal and showed that the analysis is coarse grained and has limitations in this sense. In Chapter 3 we address the third objective of this thesis by extending the study of pseudo-tested methods and evaluating their utility for developers.

CHAPTER 3

A COMPREHENSIVE STUDY OF PSEUDO-TESTED METHODS

Extreme transformations work at the global method level. This is a good granularity for developers to reason about the interaction between the application code and the test cases. If we can find pseudo-tested methods in well tested real open-source projects and they point at relevant testing issues, we hypothesize that they can serve as a solid basis to address the main objective of this thesis: provide feedback to developers about the quality of their test suite and help them improve it.

The third objective of this thesis is precisely to know to what extent pseudo-tested methods can inform developers about the quality of their tests. We want to know whether these methods are relevant indicators for developers who wish to improve their test suite. In fact, these methods may encapsulate behaviors that are poorly specified by the test suite, but are not relevant functionalities for the project.

To achieve this objective, we first challenge the external validity of Niedermayr *et al*'s experiment with new study subjects, second we perform an in-depth qualitative empirical study of pseudo-tested methods. While it seems to be intuitively true that pseudo-tested methods are indicators of badly tested code, we aim at quantifying this phenomenon. To investigate pseudo-tested methods, we perform an empirical study based on the analysis of 21 open source Java projects. In total, we analyze 28K+ methods in these projects. We articulate this chapter around three parts.

In the first part of the chapter we characterize our study subjects by looking at the number and proportion of pseudo-tested methods. This also acts as a conceptual replication [68] of Niedermayr's study. Our results mitigate two threats to the validity of Niedermayr's results: our methodology mitigates internal threats, by using another tool to detect pseudo-tested methods, and our experiment mitigates external threats by augmenting Niedermayr's original set of study objects with 19 additional open source project.

In the second part, we quantify the difference between pseudo-tested methods and the other covered methods. We compare both sets of methods with respect to the fault detection ratio of the test suite and prove that pseudo-tested methods are significantly worse tested with respect to this criterion.

In the third part, we aim at collecting a qualitative feedback from the developers. First, we manually sample a set of pseudo-tested methods that reveal specific issues in the test suites of seven projects. Then, we submit pull requests and send emails to the developers, asking their feedback about the relevance of these test issues. All pull requests have been merged to improve the test suite. Second, we met with the developers of three projects and inspected together a sample of 101 pseudo-tested methods. We found that developers consider only 30 of them worth spending time improving the test suite.

To summarize, the contributions of this chapter are as follows:

- a conceptual replication of Niedermayr's initial study on pseudo-tested methods. Our replication confirms the generalized presence of such methods, and improves the external validity of this empirical observation.
- a quantitative analysis of pseudo-tested methods, which measures how different they are compared to other covered, not pseudo-tested methods.
- a qualitative manual analysis of 101 pseudo-tested methods, involving developers, that reveals that less than 30% of these methods are clearly worth the additional testing effort.
- open science, with a complete replication package available at: `https://github.com/STAMP-project/descartes-experiments/`.

The rest of this chapter is organized as follows. Section 3.1 defines the key concepts that form the core of this empirical study. Section 3.2 introduces the research questions that we investigate in this chapter, as well as the set of study subjects and the metrics that we collect. In Section 3.3, we present and discuss the observations for each research question. In Section 3.4, we discuss the threats to the validity of this study.

3.1 Pseudo-tested Methods

Niedermayr *et al* defined pseudo-tested methods as those executed by the test suite and *none of the covering test cases fails when the whole logic of the method is removed*. In this section we derive the definition of pseudo-tested methods from the concepts discussed in Chapter 1. Our alternative definition is elaborated by considering the *effects* of the method and it is equivalent to the one given by Niedermayr. We also discuss the procedure to discover such methods in a program.

The effects of a method are the changes in the program state it produces: changes to the state of receiver instance, changes to the state of other objects (by calling other methods), return a value as a result of its computation, changes to the state of the parameters, and any other global change that the code of the method provokes *i.e.* changing the value of a global static field.

For the sake of simplicity and as introduced in Section 4.2.1, we consider a program P to be a set of methods, and a test suite T to be a set of test cases as defined in Definition 1.

Let m be a method; $S = \cup_{m \in P} \mathit{effects}(m)$ the set of effects of all methods in P; $\mathit{effects}(m)$ a function $\mathit{effects} : P \rightarrow S$ that returns all the effects of a method m; detect, a predicate $T \times S \rightarrow \{\top, \bot\}$ that determines if an effect is detected by T.

We can define pseudo-tested methods as follows.

Definition 3. *A method is said to be pseudo-tested with respect to a test suite, if the test suite covers the method and does not assess any of its effects:*

$$\forall s \in \mathit{effects}(m), \nexists t \in T : \mathit{detect}(t, s)$$

Those methods that are not pseudo-tested are considered as *required*.

Definition 4. *A method is said to be required if the test suite covers the method and assesses at least one of its effects:*

$$\exists s \in \mathit{effects}(m), \exists t \in T : \mathit{detect}(t, s)$$

Recall Listing 1.1 and Listing 1.2. In the `VersionedSet` class, the `incrementVersion` method (line 13) is pseudo-tested. The `testAdd` test case in Listing 1.2 (line 4) triggers the execution of the method but does not assess its effect: the modification of the `version` field. None of the other test cases in this test suite assess this effect. So, the body of this method could be removed and the test suite will not notice the change. This particular example also shows that pseudo-tested methods may pose a testing challenge derived from testability issues in the code. The method in question is private and its effects are produced over private fields.

One can note that Niedermayr et al. [51] call required methods, "tested methods". We do not keep this terminology here, for two key reasons: (i) these covered methods may include behaviors that are not specified by the test suite, hence not all the effects have been tested; (ii) meanwhile, by contrast to pseudo-tested methods, the correct execution of these methods is *required* to make the whole test suite pass correctly since at least one effect of the method is assessed.

3.1.1 Finding Pseudo-tested Methods

A "pseudo-tested" method, as defined previously, is an idealized concept. In this section, we describe an algorithm that implements a practical way of collecting a set of pseudo-tested methods in a program P, in the context of the test suite T, based on the original proposal of Niedermayr et al. [51]. It relies on the idea of "extreme code transformations", which consists in completely stripping out the body of a method.

Algorithm 2 starts by analyzing all methods of P that are covered by the test suite and fulfill a predefined selection criterion (predicate $\mathrm{ofInterest}$ in line 1). This criterion is based on the structure of the method and aims at reducing the number of false positives detected by the procedure.

```
   Data: P, T
   Result: pseudo: {pseudo-tested methods in P}
 1 foreach m ∈ P|covered(m, T) ∧ ofInterest(m) do
 2 |  variants : {extreme variants of m}
 3 |  if returnsValue(m) then
 4 |  |  stripBody(m)
 5 |  |  checkReturnType(m)
 6 |  |  variants ← fixReturnValues(m)
 7 |  end
 8 |  else
 9 |  |  variants ← stripBody(m)
10 |  end
11 |  failure ← false
12 |  foreach v ∈ variants do
13 |  |  P′ ← replace(m, v, P)
14 |  |  failure ← failure ∨ run(T, P′)
15 |  end
16 |  if ¬failure then
17 |  |  pseudo ← pseudo ∪ m
18 |  end
19 end
20 return pseudo
```

Algorithm 2: Procedure to detect pseudo-tested methods

Table 3.1: Extreme transformations used depending on return type.

Method type	Values used
void	-
Reference types	null
boolean	true,false
byte,short,int,long	0,1
float,double	0.0,0.1
char	‘ ’, ‘A’
String	“”, “A”
T[]	new T[]{}

It eliminates uninteresting methods such as trivial setter and getters or empty void methods. More insight on this will be given in Section 3.2.3. If the method returns a value, the body of the method is stripped out and we generate a few variants that simply return predefined values (line 3). These values depend on the return type, and are shown in Table 3.1. Compared to Niedermayr et al. [51], we add two new transformations, one to return `null` and another to return an empty array. These additions allow to expand the scope of methods to be analyzed. These are the same transformations we used for the empirical study in Chapter 2.

If the method is void, we strip the body without further action (line 8). Once we have a set of variants, we run the test suite on each of them, if no test case fails on any of the variants of a given method, we consider the method as pseudo-tested (line 16). One can notice in line 13 that all extreme transformations are applied to the original program and are analyzed separately.

Algorithm 2 has been implemented in Descartes, which was presented in Chapter 2

3.2 Experimental Protocol

Pseudo-tested methods are intriguing. They are covered by the test suite, their body can be drastically altered and yet the test suite does not notice the transformation. We design an experimental protocol to explore the nature of those methods.

3.2.1 Research Questions

Our work in this chapter is organized around the following research questions:

RQ1 *How frequent are pseudo-tested methods?*

This first question aims at characterizing the prevalence of pseudo-tested methods. It is a conceptual replication of the work by Niedermayr et al. [51], with a larger set of study objects and a different tool support for the detection of pseudo-tested methods.

RQ2 *Are pseudo-tested methods the weakest points in the program, with respect to the test suite?*

This second question aims at determining to what extent pseudo-tested methods actually capture areas in the code that are less tested than other parts. Here we use the mutation score as a standard test quality assessment metric. We compare the method-level mutation score of pseudo-tested methods against that of other covered and required methods (that are not pseudo-tested). Here by method-level mutation score we mean the mutation score computed for each method, considering only the mutants created inside each particular method.

RQ3 *Are pseudo-tested methods helpful for developers to improve the quality of the test suite?*

In this question we manually identify eight issues in the test suites of seven projects. We communicate these issues to the development teams through pull requests or email and collect their feedback.

RQ4 *Which pseudo-tested methods do developers consider worth an additional testing action?*

Following our exchange with the developers, we expand the qualitative analysis to a sample of 101 pseudo-tested methods distributed across three of our study subjects. We consulted developers to characterize the pseudo-tested methods that are worth an additional testing action and the ones that are not worth it.

3.2.2 Study Subjects

We selected 21 open source projects in a systematic manner to conduct our experiments. We considered active projects written in Java, that use Maven as main build system, JUnit as the main testing framework and their code is available in a version control hosting service, mostly Github. This is the same set of projects used in the study presented in Chapter 2. Here we provide more details about their selection as study subjects.

A project is selected if it meets one of the following conditions: (i) they are present in the experiment of Niedermayr et al. [51] (4 projects), (ii) they are maintained by industry partners from whom we can get qualitative feedback (4 projects), (iii) they are regularly used in the software testing literature (11 projects), (iv) they have a mature history with more than 12,000 commits (one project) or they have a code base surpassing one million lines of code (one project).

Table 3.2 shows the entire list. The first two columns show the name of each project and the identifiers we use to distinguish them in the present study. The third and fourth columns of this table show the number of lines of code in the main code base and testing code respectively. Both numbers were obtained using *cloc*[1]. The last column shows the number of test cases as reported by Maven when running *mvn test*. For instance, Apache Commons IO (`commons-io`) has 8 839 lines of application code and 3 463 lines of test code spread over 634 test cases. The smallest project, Apache Commons Cli (`commons-cli`), has 2 764 lines, while the largest, Amazon Web Services (`aws-sdk-java`) is composed of 1.6 million lines of Java code. The list shows that our inclusion criteria enable us to have a wide diversity in terms of project size. In many cases the test code is as large or larger than the application code base.

Appendix A contains the links to access the source code of all the projects and the identifiers of the commits marking the revision used in our study.

[1] `https://github.com/AlDanial/cloc`

Table 3.2: Projects used as study subjects. ▲: Projects taken from the work of Niedermayr et al. [51]. ▼: Projects taken from our industry partners. ⧫: Other projects used in the software testing literature. ■: Project from Github with more than 12,000 commits. ★: Projects with more than a million LOC

Project	ID	App LOC	Test LOC	Tests
▼ AuthZForce PDP Core	`authzforce`	12 596	3 463	634
★ Amazon Web Services SDK	`aws-sdk-java`	1 676 098	24 115	1 291
⧫ Apache Commons CLI	`commons-cli`	2 764	4 241	460
⧫ Apache Commons Codec	`commons-codec`	6 485	10 782	663
▲ Apache Commons Collections	`commons-collections`	23 713	27 919	13 677
⧫ Apache Commons IO	`commons-io`	8 839	15 495	963
▲ Apache Commons Lang	`commons-lang`	23 496	37 237	2 358
⧫ Apache Flink	`flink-core`	46 390	30 049	2 341
⧫ Google Gson	`gson`	7 184	12 884	951
⧫ Jaxen XPath Engine	`jaxen`	12 467	8 579	716
▲ JFreeChart	`jfreechart`	94 478	39 875	2 138
⧫ Java Git	`jgit`	75 893	52 981	2 760
⧫ Joda-Time	`joda-time`	28 724	55 311	4 207
⧫ JOpt Simple	`jopt-simple`	2 386	6 828	817
⧫ jsoup	`jsoup`	11 528	6 311	561
▼ SAT4J Core	`sat4j-core`	18 310	8 091	710
⧫ Apache PdfBox	`pdfbox`	121 121	15 978	1 519
■ SCIFIO	`scifio`	49 005	6 342	1 021
▼ Spoon	`spoon`	48 363	32 833	1 371
▲ Urban Airship Client Library	`urbanairship`	25 260	15 625	701
▼ XWiki Rendering Engine	`xwiki-rendering`	37 571	9 276	2 247
Total		2 332 671	424 215	42 106

3.2.3 Metrics

In this section, we define the metrics we used to perform the quantitative analysis of pseudo-tested methods.

Number of methods (#METH). The total number of methods found in a project, after excluding constructors and static initializers. We make this choice because these methods cannot be targeted by our extreme transformations.

Certain types of methods are not generally targeted by developers in unit tests, we exclude them to reduce the number of methods that developers may consider as false positives. In our analysis, we do not consider:

- methods that are not covered by the test suite. We ignore these methods, since, by definition, pseudo-tested methods are covered.
- `hashCode` methods, as suggested by Niedermayr et al. [51], since this type of transformation would still convey with the hash code protocol
- methods with the structure of a simple getter or setter (*i.e.* methods that only return the value of an instance field or assign a given value to an instance field), methods marked as deprecated (*i.e.* methods explicitly marked with the (@Deprecated) annotation or declared in a class marked with that same annotation), empty void methods, methods returning a literal constant and compiler generated methods such as synthetic methods or methods generated to support *enum* types

These methods are a subset of the stop-methods presented in Section 2.1.2. There we provided a comprehensive list of these methods and how they are detected by Descartes.

Number of methods under analysis (#MUA). Given a program P, that includes #METH methods, the number of methods under analysis #MUA is obtained after excluding the methods described above.

Ratio of covered methods (C_RATE). A program P can be seen, in a simplified view, as a set of methods. When running a test suite T on P a subset of methods $COV \subset P$ are covered by the test suite. The ratio of covered methods is defined as C_RATE= $\frac{|COV|}{|P|}$. In practice, COV is computed by Descartes, as explained in Section 3.1.1.

Ratio of pseudo-tested methods (PS_RATE). For all methods under analysis, we run the procedure described in Algorithm 2. This produces the subset of pseudo-tested methods, noted as #PSEUDO methods. The ratio of pseudo-tested methods in a program is computed against the methods under analysis and it is defined as PS_RATE=$\frac{\#\text{PSEUDO}}{\#\text{MUA}}$.

PS_RATE is used in RQ1 to determine the presence of pseudo-tested methods in our study subjects. We also use the Pearson coefficient to check if we can state a correlation between PS_RATE and C_RATE.

Mutation score. Given a program P, its test suite T and a set of mutation operators op, a mutation tool generates a set M of mutants for P. The mutation score of the test suite T over M is defined as the ratio of detected mutants included in M. That is:

$$score(M) = \frac{|\mu : \mu \in M \wedge detected(\mu)|}{|M|} \tag{3.1}$$

where $detected(\mu)$ means that at least one test case in T fails when the test suite is executed against μ.

Mutation score for pseudo-tested methods (MS_pseudo): the score, as defined in Equation 3.1, but M is the subset of mutants generated on the pseudo-tested methods of P.

Mutation score for required methods (MS_req): the score, as defined in Equation 3.1 but M is the subset of mutants generated only on the required methods of P.

These three mutation score metrics are used in RQ2 to quantitatively determine if pseudo-tested methods are the worst tested methods in the code base. We perform a Wilcoxon statistical test to compare the values obtained for MS_pseudo and MS_req on our study subjects.[2]

3.3 Experimental Results

The following sections present in depth our experimental results.

3.3.1 RQ1: How frequent are pseudo-tested methods?

We analyzed each study subject following the procedure described in Section 3.1.1. The results are summarized in Table 3.3. The second column shows the total number of methods excluding constructors. The third, lists the methods covered by the test suite. The following column shows the ratio of covered methods. The “#MUA” column shows the number of methods under analysis, per the criteria described in Section 3.2.3. The last two columns give the number of pseudo-tested methods (#PSEUDO) and their ratio to the methods under analysis (PS_RATE). In Figure 3.1 we represent PS_RATE by showing #PSEUDO in proportion to #MUA for each project.

For instance, in `authzforce`, 325 methods are covered by the test suite, of which 291 are relevant for the analysis. In total, we identify 13 pseudo-tested methods representing 4% of the methods under analysis.

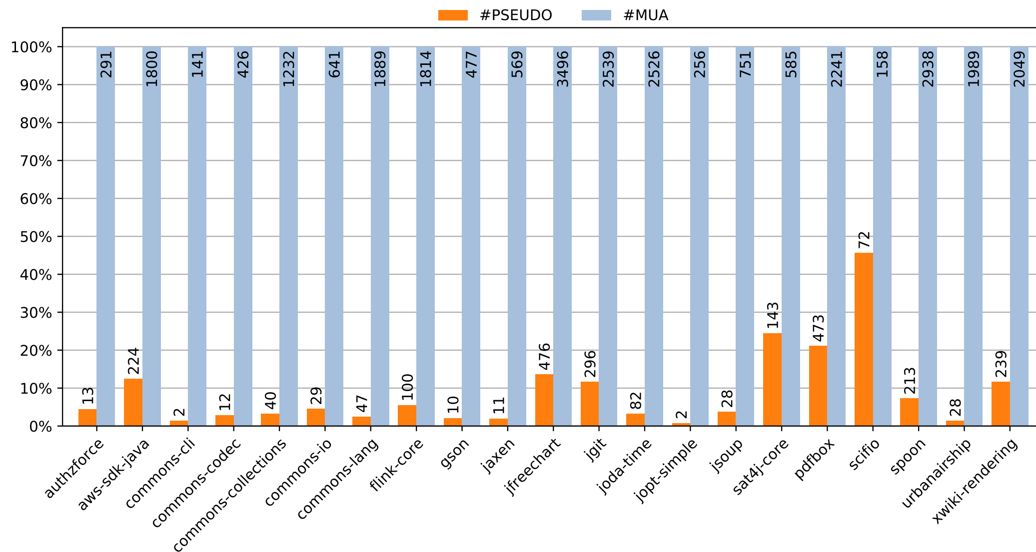
We discover pseudo-tested methods in all our study objects, even for those with high coverage. This corroborates the observations made by Niedermayr et al. [51]. The number of observed pseudo-tested methods ranges from 2 methods, in `commons-cli`, to 476 methods in `jfreechart`. The PS_RATE varies from 1% to 46%. In 14 cases its value remains below 7% of all analyzed methods. This means that, compared to the total number of methods in a project, the amount of pseudo-tested methods can be managed by the developers in order to guide the improvement of test suites.

Analysis of outliers

Some projects have specific features that may affect the PS_RATE value in a distinctive way. In this section we discuss those cases.

[2]The computation of the Pearson coefficient and the Wilcoxon test were performed using the features of the R language.

Figure 3.1: Proportion of pseudo-tested methods with respect of all methods under analysis for each each project



authzforce, uses almost exclusively parameterized tests. The ratio of covered methods is low, but these methods have been tested exhaustively and therefore they have a lower PS_RATE compared to other projects with similar coverage.

The application domain can have an impact on the design of test suites. For example, scifio is a framework that provides input/output functionalities for image formats that are typically used in scientific research. This project shows the highest PS_RATE in our study. When we look into the details, we find that 62 out of their 72 pseudo-tested methods belong to the same class and deal with the insertion of metadata values in DICOM images, a format widely used in medical imaging. Not all the metadata values are always required and the test cases covering these methods do not check their presence. pdfbox is another interesting example in this sense. It is a library designed for the creation and manipulation of PDF files. Some of their functionalities can only be checked by visual means which increases the level of difficulty to specify an automated and fine-grained oracle. Consequently, this project has a high PS_RATE.

At the other end of the PS_RATE spectrum, we find commons-cli and jopt-simple. These are small projects, similar in purpose and both have comprehensive test suites that reach 97% and 98% of line coverage respectively (as measured by *cobertura*[3]). Only two pseudo-tested methods were found for each one of them. Three of those four methods create and return an exception message. The remaining method is a toString implementation.

Relationship between pseudo-tested methods and coverage

We observe that the projects with lowest method coverage show higher ratios of pseudo-tested methods. The Pearson coefficient between the coverage ratio and the ratio of pseudo-tested methods is -0.67 and $p < 0.01$ which indicates a moderate negative relationship.

This confirms our intuition that pseudo-tested methods are more frequent in projects that are poorly tested (high pseudo-tested ratios and low coverage ratios). However, the ratio of pseudo-tested methods is more directly impacted by the way the methods are verified and not the ratio of methods covered. It is possible to achieve a low ratio of pseudo-tested methods covering a small portion of the code. For example, authzforce and xwiki-rendering have comparable coverage ratios but the former has a lower ratio of pseudo-tested methods. The correlation with the ratio is a consequence of the fact that, in general, well tested projects also have higher coverage ratios.

[3] http://cobertura.github.io/cobertura/

Table 3.3: Number of methods in each project, number of methods under analysis and number of pseudo-tested methods

Project	#Methods	#Covered	C_RATE	#MUA	#PSEUDO	PS_RATE
authzforce	697	325	47%	291	13	4%
aws-sdk-java	177 449	2 314	1%	1 800	224	12%
commons-cli	237	181	76%	141	2	1%
commons-codec	536	449	84%	426	12	3%
commons-collections	2 729	1 270	47%	1 232	40	3%
commons-io	875	664	76%	641	29	5%
commons-lang	2 421	1 939	80%	1 889	47	2%
flink-core	4 133	1 886	46%	1 814	100	6%
gson	624	499	80%	477	10	2%
jaxen	958	616	64%	569	11	2%
jfreechart	7 289	3 639	50%	3 496	476	14%
jgit	6 137	3 702	60%	2 539	296	12%
joda-time	3 374	2 783	82%	2 526	82	3%
jopt-simple	298	265	89%	256	2	1%
jsoup	1 110	844	76%	751	28	4%
sat4j-core	2 218	613	28%	585	143	24%
pdfbox	8 164	2 418	30%	2 241	473	21%
scifio	3 269	895	27%	158	72	46%
spoon	4 470	2 976	67%	2 938	213	7%
urbanairship	2 933	2 140	73%	1 989	28	1%
xwiki-rendering	5 002	2 232	45%	2 049	239	12%
Total	234 923	32 650	14%	28 808	2 540	9%

Comparison with the study by Niedermayr et al. [51]

Our study, on a new dataset, confirms the major finding of Niedermayr et al. [51]'s study: pseudo-tested methods exist in all projects, even the very well tested ones. This first-ever replication improves the external validity of this finding. We note in the original study by Niedermayr et al. [51], that the reported ratio was higher, ranging from 6% to 53%. The difference can be explained from the fact that (i) we exclude deprecated methods and (ii) we consider two new other mutation operators. These two factors change the set of methods that have been targeted.

Answer to RQ1

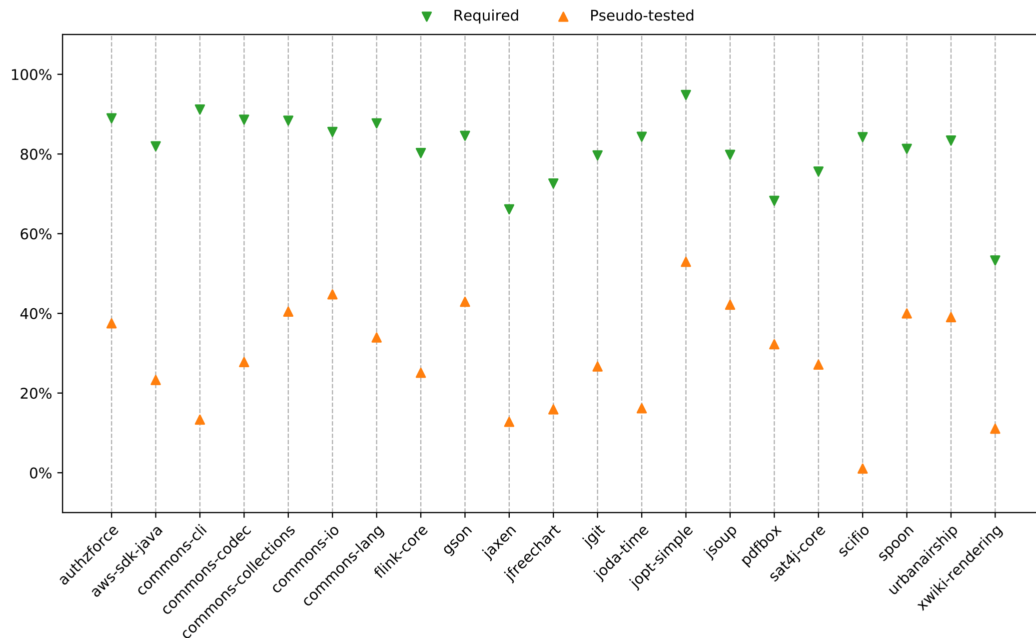
We have made the first independent replication of Niedermayr *et al*'s study. Our replication confirms that all Java projects contain pseudo-tested methods, even the very well tested ones. This improves the external validity of this empirical fact. The ratio of pseudo-tested methods with respect to analyzed methods ranged from 1% to 46% in our dataset.

3.3.2 RQ2: Are pseudo-tested methods the weakest points in the program, with respect to the test suite?

By definition, test suites fail to assess the presence of any effect in pseudo-tested methods. As such, these methods can be considered as very badly tested, even though they are covered by the test suite. To further confirm this fact we assess the test quality of these methods with a traditional test adequacy criterion: mutation testing [22]. To do so, we measure the chance for a mutant planted in a pseudo-tested method to be detected (killed).

For each of our study subjects, we run a mutation analysis based on PITest, a state-of-the-art mutation tool for Java. We configure PITest with its standard set of mutation operators. PITest is capable of listing: the comprehensive set of mutants, the method in which they have been inserted

Figure 3.2: Mutation score for mutants placed inside pseudo-tested and required methods.



and whether they have been detected (killed) by the test suite. We extract the set of mutants that have been placed in the body of the pseudo-tested methods to compute the mutation score on those methods (MS_pseudo) as well as the mutation score of required methods (MS_req).

Figure 3.2 shows the results of this experiment. In all cases, the mutation score of pseudo-tested methods is significantly lower than the score of normal required methods. This means that a mutant planted inside a pseudo-tested method has more chances to survive than a mutant planted in required methods. The minimum gap is achieved in `pdfbox` with scores 32% for pseudo-tested methods and 68% for others. For `scifio`, only 1% of PITest mutants in pseudo-tested methods can be killed (as opposed to 84% in required methods). To validate this graphical finding, we compare MS_pseudo and MS_req.

In average the mutation score of required methods is 52% above that of pseudo-tested methods. With the Wilcoxon statistical test, this is a significant evidence of a difference with a p-value $p < 0.01$. The effect size is 1.5 which is considered as large per the standard guidelines in software engineering [39].

Analysis of interesting examples

It calls the attention, that, in no case, MS_pseudo was 0%. So, even when extreme transformations are not spotted by the test suite, some mutants inside these methods can be detected. We now explain this case.

Listing 3.1 shows a simplified extract of a pseudo-tested method we have found in `auzthforce` and where some traditional mutants were detected. The `checkNumberOfArgs` method is covered by six test cases and was found to be pseudo-tested. In all test cases, the value of `numInputs` is greater than two, hence the condition on line 3 was always false and the exception was never thrown. PITest created five mutants in the body of this method and two of them were detected. Those mutants replaced the condition by $true$ and $<$ by $<=$ respectively. With this, the condition is always $false$, the exception is thrown and the mutants are detected. It means that those mutants are trivially detected with an exception, not by an assertion. This is the major reason for which the mutation score of pseudo-tested methods can be higher than 0.

This explanation holds for `jopt-simple` which achieves a seemingly high 53% MS_pseudo. A total of 17 mutants are generated in the two pseudo-tested methods of the project. Nine of these mutants are killed by the test suite. From these nine, six replaced internal method calls by a default value and the other three replaced a constant by a different value. All nine mutations made the program crash with an exception, and are thus trivially detected.

```java
class AnyOfAny {
  protected void checkNumberOfArgs(int numInputs) {
    if(numInputs < 2)
      throw new IllegalArgumentException();
  }

  public void evaluate(... args) {
    checkNumberOfArgs(args.size())
    ...
  }
}
```

Listing 3.1: A pseudo-tested method where traditional mutants were detected

`scifio` has the lowest MS_pseudo. PITest generated 7 598 mutants in the 62 methods dealing with metadata and mentioned in Section 3.3.1. The mutants modify the metadata values to be placed in the image and, as discussed earlier, those values are not specified in any oracle of the test suite. Hence, none of these mutants are detected.

Distribution of method-level mutation score

To further explore the difference between MS_pseudo and MS_req, we compute the distribution of the method-level mutation score. That is, we compute a mutation score for each method in a project. The final distribution for each project is shown in Figure 3.3. Each row displays two violin plots for a specific project.[4] Each curve in the plot represents the distribution of mutation scores computed per method. The thicker areas on each curve represent scores achieved by more methods.

The main finding is that the distributions for the required methods are skewed to higher mutation scores, while the scores for pseudo-tested methods tend to be skewed to lower values. This is a clear trend, which confirms the results of Figure 3.2. It is also the case that most distributions cover a wide range of values. While most pseudo-tested methods have low scores, there are cases for which the mutation score could reach high values, due to trivial exception-raising mutants.

We observe that 63 pseudo-tested methods across all projects have a 100% mutation score. Among those 63, 34 have only one or two trivial mutants. As an extreme case, in the `jsoup` project we find that the method `load` of the `Entities` class[5] is pseudo-tested and PITest generates 69 mutants that are all killed. All mutants make the program crash with an exception, yet the body of the method can be removed and the absence of effects is unnoticed by the test suite. This suggests that the extreme transformations performed to find pseudo-tested methods are less susceptible to be trivially detected.

Answer to RQ2

The hypothesis that pseudo-tested methods expose weakly tested regions of code is confirmed by mutation analysis. For all the 21 considered projects, the mutation score of pseudo-tested methods is significantly lower than the score of required methods, a finding confirmed by a very low p-value lower than 0.01 and a very high effect size of 1.5.

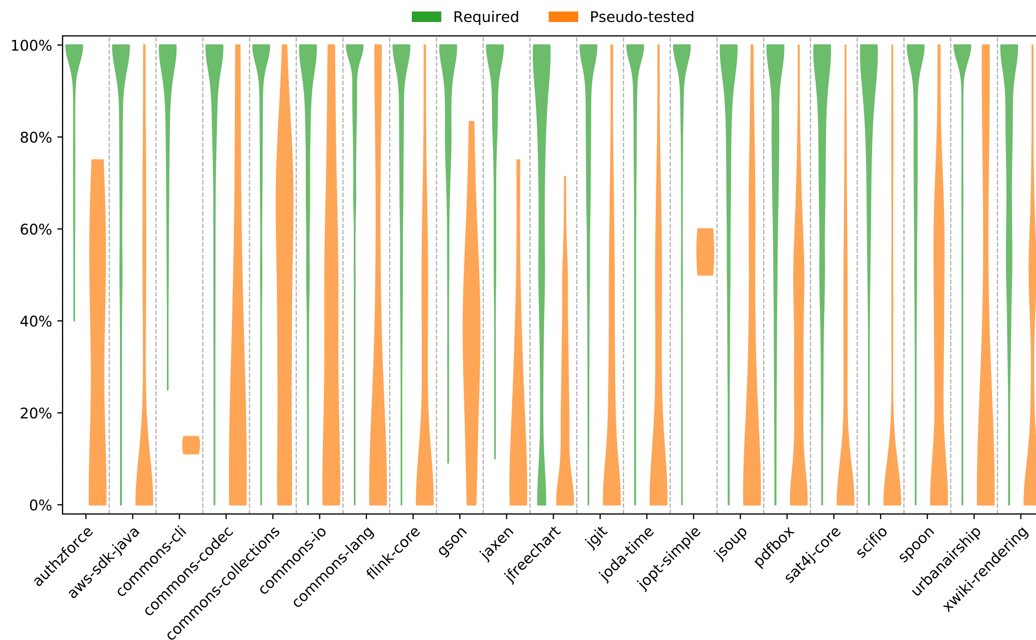
3.3.3 RQ3: Are pseudo-tested methods relevant for developers to improve the quality of the test suite?

To answer this question, we manually analyze the *void* and *boolean* pseudo-tested methods which are accessible from an existing test class. *void* and *boolean* methods have only one or two pos-

[4] The violin plot for pseudo-tested methods of `commons-cli` and `jopt-simple` are not displayed, as they have too few methods in this category.

[5] https://github.com/jhy/jsoup/blob/35e80a779b7908ddcd41a6a7df5f21b30bf999d2/src/main/java/org/jsoup/nodes/Entities.java#L295

Figure 3.3: PITest method-level mutation score distribution by project and method category



sible extreme transformations, thus are easier to explain to developers. We identify eight testing issues revealed by these pseudo-tested methods: two cases of a miss-placed oracle, two cases of missing oracle, three cases of a weak oracle and one case of a missing test input. These issues have been found in seven of our study subjects.

For each testing issue we prepare a pull request that fixes the issue, or we send the information by email. Our objective is to collect qualitative feedback from the development teams about the relevance of the testing issues revealed by pseudo-tested methods.

We now summarize the discussion about each testing issue.

Feedback from `aws-sdk-java`

Per our selection criterion, we have spotted one pseudo-tested method. We made one pull request (PR)[6] to explicitly assess the effects of one pseudo-tested method named `prepareSocket`. This method is covered by four test cases that follow the same pattern. A simplified extract is shown in Listing 3.2. The test cases mock a socket abstraction that verifies if a given array matches the expected value. `prepareSocket` should call the `setEnabledProtocols` in the socket abstraction. When running an extreme transformation on this method, the assertion is never evaluated and the test cases pass silently. In the pull request we moved the assertion out of the `setEnabledProtocols` method, in order to have it verified after `prepareSocket`. Listing 3.3 shows a simplified version of the proposed code. With this modification, the method is not pseudo-tested anymore. The developer agreed that the proposed change was an improvement and the pull request was merged into the code. This is an example of a miss-placed oracle and the value of pseudo-tested methods.

```
1  @Test
2  void typical() {
3    SdkTLSSocketFactory f = ...;
4    //prepareSocket was found to be pseudo-tested
5    f.prepareSocket(new TestSSLSocket() {
6      ...
7      @Override
8      public void setEnabledProtocols(String[] protocols) {
9      assertTrue(Arrays.equals(protocols, expected));
10     }
```

[6] https://github.com/aws/aws-sdk-java/pull/1437

```
11  ...
12   });
13 }
```

Listing 3.2: A weak test case for method prepareSocket.

```
1  @Test
2  void typical() {
3    SdkTLSSocketFactory f = ...;
4    SSLSocket s = new TestSSlSocket() {
5    @Override
6    public void setEnabledProtocols(String[] protocols) {
7      capturedProtocols = protocols;
8    }
9    ...
10   };
11   f.prepareSocket(s);
12   //This way the test fails if no protocol was enabled
13   assertArrayEquals(s.capturedProtocols, expected));
14 }
```

Listing 3.3: Proposed test improvement. The assertion was moved out of the socket implementation. Consequently, `prepareSocket` is no longer pseudo-tested

Feedback from `commons-collections`

In this project, certain methods implementing iterator operations are found to be pseudo-tested. Specifically two implementations of the `add` method and four of the `remove` method are pseudo-tested in classes where these operations are not supported. Listing 3.4 shows one the methods and a simplified extract of the test case designed to assess their effects. If the `add` method is emptied, then the exception is never thrown and the test passes. We proposed a pull request [7] with the change shown in Listing 3.5. The proposed change verifies that an exception has been thrown. As in the previous example, the issue is related to the placement of the assertion. The developer agreed to merge the proposed test improvement into the code. This is a second example of the value of pseudo-tested methods. Being in a different project, and assessed by another developer, this increases our external validity.

```
1    class SingletonListIterator
2      implements Iterator<Node> {
3      ...
4      void add() {
5      //This method was found to be pseudo-tested
6      throw new UnsupportedOperationException();
7      }
8      ...
9    }
10
11   class SingletonListIteratorTest {
12     ...
13     @Test
14     void testAdd() {
15     SingletonListIterator it = ...;
16     ...
17     try {
18       //If the method is emptied, then nothing happens
19       //and the test passes.
20       it.add(value);
21     } catch(Exception ex) {}
22     ...
23   }
```

Listing 3.4: Class containing the pseudo-tested method and the covering test class.

[7] https://github.com/apache/commons-collections/pull/36

```
...
try {
  it.add(value);
  fail(); //If this is executed,
      //then the test case fails
} catch(Exception ex) {}
...
```

Listing 3.5: Change proposed in the pull request to verify the unsupported operation.

Feedback from `commons-codec`

For `commons-codec` we found that the *boolean* method `isEncodeEqual` was pseudo-tested. The method is covered by only one test case, shown in Listing 3.6. As one can notice, the test case lacks the corresponding assertion. So, none of the extreme transformations applied to this method could cause the test to fail.

```
public void testIsEncodeEquals() {
    final String[][] data = {
    {"Meyer", "M\u00fcller"},
    {"Meyer", "Mayr"},
    ...
    {"Miyagi", "Miyako"}
  };
    for (final String[] element : data) {
            final boolean encodeEqual =
    this.getStringEncoder().isEncodeEqual(element[1], element[0]);
 }
}
```

Listing 3.6: Covering test case with no assertion.

All the inputs in the test case should make the method return *true*. When we placed the corresponding assertion we found that the first input (in line 3) was wrong and we replaced it by a correct pair of values. We made a pull request[8] and the fixture was accepted by the developers and also slightly increased the code coverage by 0.2%.

Feedback from `commons-io`

In `commons-io` we found several *void* `write` methods of the `TeeOutputStream` to be pseudo-tested. This class represents an output stream that should send the data being written to other two output streams. A reduced version of the test case covering these methods can be seen in Listing 3.7. Line 7 shows that the assertion checks that both output streams should contain the same data. If the `write` method is emptied, nothing is written to both streams but the assertion remains valid as both have the same content (both are empty). The test case should verify not only that those two streams have the same content but that they have the right value. In this sense, we say that this is an example of a weak oracle. We made a pull request[9] with the changes exposed in Listing 3.8. The change adds a third output stream to be used as a reference value. The pull request was accepted and it slightly increased the code coverage by 0.07%.

```
public void testTee() {
  ByteArrayOutputStream baos1 = new ByteArrayOutputStream();
  ByteArrayOutputStream baos2 = new ByteArrayOutputStream();
  TeeOutputStream tos = new TeeOutputStream(baos1, baos2);
  ...
  tos.write(array);
  assertByteArrayEquals(baos1.toByteArray(), baos2.toByteArray());
}
```

Listing 3.7: Test case verifying `TeeOutputStream` write methods

[8] https://github.com/apache/commons-codec/pull/13
[9] https://github.com/apache/commons-io/pull/61

```
1  public void testTee() {
2    ByteArrayOutputStream baos1 = new ByteArrayOutputStream();
3    ByteArrayOutputStream baos2 = new ByteArrayOutputStream();
4    ByteArrayOutputStream expected = new ByteArrayOutputStream();
5    TeeOutputStream tos = new TeeOutputStream(baos1, baos2);
6    ...
7    tos.write(array);
8    expected.write(array);
9    assertByteArrayEquals(expected.toByteArray(), baos1.toByteArray());
10   assertByteArrayEquals(expected.toByteArray(), baos2.toByteArray());
11 }
```

Listing 3.8: Change proposed to verify the result of the write methods

For three projects, `spoon`, `flink-core` and `sat4j-core`, we discuss the details of the testing issues[10] directly with the developers via emails. We systematically collected their feedback.

Feedback from `spoon`

We ask the project team about a public *void* method, named `visitCtAssert`[11], and covered indirectly by only one test case. This method was part of a visitor pattern implementation, which is common inside this project. This particular method handles *assert* Java expressions in an Abstract Syntax Tree. The test case does not assess the effects of the method. The developers expressed that this method should be verified by adding a stronger verification or a new test case. They were interested in our findings. They took no immediate action but opened a general issue [12].

Feedback from `flink-core`

This team was contacted to discuss about a public *void* method, named `configure`[13], which is directly called by a single test case. This particular method loads a given configuration and prevents a field value from being overwritten. Listing 3.9 shows a simplified extract of the code. The body of the method could be removed and the test passes as the assertion only involves the initial value of the field. The developers explained that the test case was designed precisely to verify that the field is not changed after the method invocation. They expressed that more tests could probably make the scenario more complete. In our view, the test case should assert both facts: the configuration being loaded and the value not being changed. The current oracle expresses a weaker condition as the former verification is not done. If the body is erased, the configuration is never loaded and the value, of course, is never changed. The developers did not take any further action.

```
1 public void configure(Configuration parameters) {
2   super.configure(parameters);
3   if(this.blockSize == NATIVE_BLOCK_SIZE) {
4     setBlockSize(...);
5   }
6 }
```

Listing 3.9: Pseudo-tested method in `flink-core`

Feedback from `sat4j-core`

We contacted the `sat4j-core` lead developer about two *void* methods. One of them, named `removeConstr`[14], was covered directly by only one test case to target a specific bug and avoid

[10] https://github.com/STAMP-project/descartes-experiments/blob/6f8a9c7c111a1da5794622652eae5327d0571ef1/direct-communications.md

[11] https://github.com/INRIA/spoon/blob/fd878bc71b73fc1da82356eaa6578f760c70f0de/src/main/java/spoon/reflect/visitor/DefaultJavaPrettyPrinter.java#L479

[12] https://github.com/INRIA/spoon/issues/1818

[13] https://github.com/apache/flink/blob/740f711c4ec9c4b7cdefd01c9f64857c345a68a1/flink-core/src/main/java/org/apache/flink/api/common/io/BinaryInputFormat.java#L86

[14] https://gitlab.ow2.org/sat4j/sat4j/blob/09e9173e400ea6c1794354ca54c36607c53391ff/org.sat4j.core/src/main/java/org/sat4j/tools/xplain/Xplain.java#L214

regression issues. The other method, named `learn`[15], was covered indirectly by 68 different test cases. The lead developer considered the first method as helpful to realize that more assertions were needed in the covering test case. Consequently, he made one commit[16] to verify the behavior of this method.

The second pseudo-tested method was considered a bigger problem, because it implements certain key optimizations for better performance. The tests cases triggered the optimization code but did not leverage the optimized result. Their result were the same with or without the optimization code. Consequently, the developer made a new commit[17] with an additional, more complex, test case where the advantages of the optimization could be witnessed.

Discussion

We now discuss the main findings of this qualitative user study.

First, all developers agreed that it is easy to understand the problems identified by pseudo-tested methods. This confirms the fact that, we, as outsiders to those projects, with no knowledge or experience, can also grasp the issue and propose a solution. The developers acknowledged the relevance of the uncovered flaws.

Second, when developers were given the solution for free (through pull requests written by us), they accepted the test improvement.

Third, when the developers were only given the problem, they did not always act by improving the test suite. They considered that pseudo-tested methods provide relevant information, and that it would make sense to enhance their test suites to tackle the issues. But they do not consider these improvements as a priority. With limited resources, the efforts are directed to the development of new features and to fix existing bugs, not to improve existing tests.

Of the eight testing issues found, seven can be linked to oracle issues and one to an input problem.

Answer to RQ3

Pseudo-tested methods uncover flaws in the test suite which are considered relevant by developers. These methods enable one to well understand the problem in a short time. However, fixing the test flaws requires some time and effort that cannot always be given, due to higher priority tasks such as new features and bug fixing.

3.3.4 RQ4: Which pseudo-tested methods do developers consider worth an additional testing action?

To answer this question we contact the development teams directly. We select three projects for which the developers have accepted to discuss with us: `authzforce`, `sat4j-core` and `spoon`. We set up a video call with the head of each development team. The goal of the call is to present and discuss a selection of pseudo-tested methods in approximately 90 minutes. With this discussion, we seek to know which pseudo-tested methods developers consider relevant enough to trigger additional work on the test suite and approximate their ratio on each project.

For projects `sat4j-core` and `spoon`, we randomly choose $\tilde{2}5$% of all pseudo-tested methods. The third project, `authzforce`, has only 13 of such methods so we consider them all, as it is a number that can be discussed in reasonable time. We prepared a report for the developers that contains the list of pseudo-tested methods, with the extreme transformations that were applied and the test cases covering the method. To facilitate the discussion we also included links to the exact version of the code we analyzed. This information was made available to the developers before the meeting.

[15] https://gitlab.ow2.org/sat4j/sat4j/blob/09e9173e400ea6c1794354ca54c36607c53391ff/org.sat4j.core/src/main/java/org/sat4j/minisat/core/Solver.java#L384
[16] https://gitlab.ow2.org/sat4j/sat4j/commit/afab137a4c1a54219f3990713b4647ff84b8bfea
[17] https://gitlab.ow2.org/sat4j/sat4j/commit/46291e4d15a654477bd17b0ce905926d24e042ca

Table 3.4: The pseudo-tested methods systematically analyzed by the lead developers, through a video call.

Project	Sample size	Worth	Percentage	Time spent (HH:MM)
authzforce[18]	13 (100%)	6	46%	29 min
sat4j-core[19]	35 (25%)	8	23%	1 hr 38 min
spoon[20]	53 (25%)	16	23%	1 hr 14 min
Total	101	30	30%	3 hr 21 min

For each method, we asked the developers to determine if: (i) given the structure of the method, and, (ii) given its role in the code base, they consider it is worth spending time creating new test cases or fixing existing ones to specify those methods. We also asked them to explain the reasons behind their decision.

Table 3.4 shows the projects involved, footnotes with links to the online summary of the interviews, the number of pseudo-tested methods included in the random sample, the number of methods worth an additional testing action and the percentage they represent with respect to the sample. We also show how much time we spent in the discussion.

We observe that only 23% of pseudo-tested methods in `sat4j-core` and `spoon`, the two largest projects, are worth additional testing actions (having the same percentage is purely coincidental). For `authzforce` the percentage of methods to be specified is 46%, but the absolute number (6) does not differ much from `sat4j-core` (8). This indicates that, potentially, many pseudo-tested come from functionalities considered less important or not a priority, therefore not well tested. The proportion of methods considered as worthless additional testing appears surprisingly high. It is important to notice that, among pseudo-tested methods, developers find cases, in their own words, “surprising” and “definitively not well tested, but they should be”. Even for the cases they don’t consider important, a developer from `sat4j-core` state that they “would like to know in which scenarios the transformation was discovered”.

We now enumerate the main reasons given by developers to consider a method worth or worthless spending time creating specific testing actions. We also include the projects in which these reason manifested.

Worthless specifying: A pseudo-tested method could be considered as useless to test, *i.e.*, *not important*, if it meets one of the following criteria:

- The code has been automatically generated (`spoon`).
- The method is part of debug functionalities, *i.e.*, formatting a debug or log message or creating and returning an exception message or an exception object (`authzforce`).
- The method is part of features that are not widely used in the code base or in external client code (`authzforce`, `sat4j-core`, `spoon`).
- The method has not been deprecated but its functionality is being migrated to another interface (`spoon`).
- The code of the method is considered as simple or trivial to need a specific test case (`sat4j-core`, `spoon`). Short methods, involving a few simple instructions are generally not considered worth to be specified by direct unit test cases (`spoon`). Listing 3.11 shows two examples that `spoon` developers consider too simple to be worth of additional testing actions.
- Methods created just to complete an interface implementation (`spoon`). The object oriented design may involve classes that need to implement a given interface but do not actually need to provide a behavior for all methods. In those cases, developers write a placeholder body which they are not interested in testing.

[18] https://github.com/STAMP-project/descartes-experiments/blob/master/actionable-hints/authzforce-core/sample.md

[19] https://github.com/STAMP-project/descartes-experiments/blob/master/actionable-hints/sat4j-core/sample.md

[20] https://github.com/STAMP-project/descartes-experiments/blob/master/actionable-hints/spoon/sample.md

```
...
public void addArrayReference(CtArrayTypeReference<?> typeReference) {
    arrayTypeReference.setComponentType(typeReference);
}
...
```

Listing 3.10: Pseudo-tested method involving a delegation pattern.

```
...
public void externalState() {
  this.selectedState = external;
}
...

public boolean matches(CtElement e) {
  e.setFactory(f);
  return false;
}
```

Listing 3.11: Pseudo-tested methods considered as too simple to require more testing actions.

- Receiving methods in a delegation pattern that add little or no logic when invoking the delegate (`spoon`). The delegation pattern exposes a method that simply calls another method (delegate). Delegate methods may have the same signature as the receiving method. The receiving method usually adds no or very little custom logic (e.g., provide a default value for unused parameters or process the returning value). Listing 3.10 shows an example of this pattern that developers do not consider to be worth of additional testing actions. If the delegate is pseudo-tested then the receiving method will be pseudo-tested as well. The opposite does not have to be necessarily true. In any case, the method exposing the actual functionality to be tested is the delegate. The receiving method may not have the same importance.

Worth specifying with additional tests: On the other hand, developers provided the following reasons when they consider a pseudo-tested method to be worth of additional testing actions:

- A method that supports a core functionality of the project or part of the main responsibility of the declaring class (`authzforce`, `sat4j-core`, `spoon`). For example, we find a class named `VisitorPartialEvaluator` which implements a visitor pattern over a Java program Abstract Syntax Tree (AST). This class simplifies the AST by evaluating all expressions that could be statically reduced. The method `visitCtAssignment`[21], declared in this class, handles assignment instructions and was found to be pseudo-tested. Assignments may influence the evaluation result, so this method plays an important role in the class.

- A method supporting a functionality that is widely used in the code base. It could be the method itself that is being frequently used or the class that declares the method (`authzforce`, `sat4j-core`, `spoon`).

- A method known to be relevant for external client code (`sat4j-core`).

- A new feature that is partially supported or not completed yet, which requires a clear specification (`spoon`).

- Methods which are the only possible way to access certain features of a class (`authzforce`). For example, a public method that calls several private methods which actually contain the implementation of the public behavior.

- Methods verifying preconditions (`authzforce`). These methods guarantee the integrity of the operations to be performed. Listing 3.12 shows an example of one of those methods consid-

[21] https://github.com/INRIA/spoon/blob/fd878bc71b73fc1da82356eaa6578f760c70f0de/src/main/java/spoon/support/reflect/eval/VisitorPartialEvaluator.java#L515

```java
protected final void checkNumberOfArgs(final int numInputs)
{
  if (numInputs != 3)
  {
    throw new IllegalArgumentException(...);
  }
}
```

Listing 3.12: A simple method that checks a precondition

ered to be important to specify. Despite the simplicity of the implementation, the `authzforce` developers consider that it is important to specify them as accurately as possible.

We have observed cases where a method meets criteria to be worth of specification and at the same time to be worthless of additional testing actions. The final decision of developers in those cases is subjective and responds to their internal knowledge about the code base. This means that it is difficult to devise an automatic procedure able to automatically determine which methods are worth of additional testing actions.

Answer to RQ4

In a sample of 101 pseudo-tested methods, systematically analyzed by the lead developers of 3 mature projects, 30 methods (30%) were considered worth of additional testing actions. The developer decisions are based on a deep understanding of the application domain and design of the application. This means that it is not reasonable to prescribe the absolute absence (zero) of pseudo-tested methods.

3.4 Threats to validity

RQ1 and RQ2. A threat to the quantitative analysis of RQ1 and RQ2 relates to external validity:

- Some extreme transformations could generate programs that are equivalent to the original. Given the nature of these transformations many of possible equivalent variants are detected by inspecting the method before applying the transformation. Methods with empty body and those returning a constant value, are skipped from the analysis. This is a problem extreme transformations have in common with traditional mutation testing. The equivalent mutant problem could also affect the value of the mutation scores computed to answer RQ2.
- The values used to transform the body of non-void and non-boolean methods may affect their categorization as pseudo-tested or required. A different set of values may produce a different categorization. Since only one detected value is needed to label a method as required, then we actually produce an over-estimation of these methods. More values could be used to reduce the final set. In our study we find only 916 of non-void and non-boolean pseudo-tested methods which represents a 36% of the total number of methods. $void$ and $boolean$ methods tend to produce more pseudo-tested methods in our study subjects.
- As pointed by Petrovic and Ivankovic [59], mutation testing results can be affected by the programming language. This affects both, the extreme transformations performed and the mutation testing validation in RQ2. All our study subjects are Java projects, so our findings can not be generalized to other languages.
- We have considered all test cases equally and did not attempt to distinguish between unit and integration tests. We made this decision because such a distinction would require setting an arbitrary threshold above which a JUnit test case is considered an integration test. Yet, considering all test cases equally could influence the amount of pseudo-tested methods.

RQ3 and RQ4. The outcome of the qualitative analysis is influenced by our insight into each project, the insight of the developers consulted, the characteristics of each code base and the methods presented. Some of the teams showed more interest and gave more importance to the findings than others. This was expected. Not all developers had strong opinions regarding the presented issues.

3.5 Conclusion

In this chapter, we provide an in-depth analysis of pseudo-tested methods found in open source projects. These methods, first coined by Niedermayr et al. [51], are intriguing from the perspective of the interaction between a program and its test suite: their code is executed when running the test suite, yet, none of their effects are assessed by the test cases. We study whether these methods are an indicator of the quality of the test suite, so we can leverage them to fulfill our main objective: help developers improve their test cases.

The novelty of our contribution with respect to the paper of Niedermayr and colleagues is three-fold. First, we perform a study with novel study objects (19 among the 21 projects studied here are not analyzed by Niedermayr in his original paper) and a different tool to detect the pseudo-tested methods. This mitigates both internal and external threats to the validity of Niedermayr's results. Second, we perform a novel study about the adequacy of the test suite for pseudo-tested methods. Third, our most significant novel contribution consists in extensive exchanges and interactions with software developers to understand the type of testing issues that are revealed by pseudo-tested methods, as well as the characteristic of pseudo-tested methods that developers consider worth an additional testing effort.

The key findings discussed in this chapter are as intriguing as the original concept of pseudo-tested methods. First, we observe that all 21 mature Java projects that we study include such methods: from 1% to 46% in our dataset. Second, we confirm that pseudo-tested methods are poorly tested, compared to the required methods: the mutation score of the former is systematically and significantly lower than the score of the latter. Third, our in-depth qualitative analysis of pseudo-tested methods and feedback from developers reveals the following facts:

- We assessed the relevance of pseudo-tested methods as concrete hints to reveal weak test oracles. These issues in the suite were confirmed by the developers, who accepted the pull requests that we proposed, to fix weak oracles.
- Less than 30% of pseudo-tested methods in a sample of 101 represent an actual hint for further actions to improve the test suite. Among the 70% considered as worthless an additional testing action we found methods not widely used in the code base, automatically generated code, trivial methods, helper methods for debugging and receiving methods in delegation patterns.
- The pseudo-tested methods that actually reveal an issue in the interaction between the program and its test suite are involved in core functionalities, are widely used in the code base, are used by external clients or verify preconditions on input data.

As discussed in Section 1.2.1 the literature [26] recognizes that understanding why a traditional mutant is not detected by a test suite is a very hard task. Our work and our exchanges with developers show that undetected extreme transformations are easier to grasp, yet still challenging in many cases. Understanding why the test suite is not able to detect such transformations involves a deep comprehension of the interplay between the test cases and the method that has been transformed.

In the next chapter, we propose an automatic technique that produces concrete clues on why an extreme transformation is not detected and how developers can deal with those testing issues. We also explore what current state-of-the-art test generation techniques can do against undetected extreme transformations. This shall address the fourth and last objective of this thesis.

CHAPTER 4

SUGGESTIONS ON TEST SUITE IMPROVEMENTS WITH AUTOMATIC INFECTION AND PROPAGATION ANALYSIS

In Chapter 3 we addressed the third objective of this thesis by demonstrating two key phenomena: (i) the pseudo-tested methods, those where an extreme transformation is not detected, are indeed the worst tested in the program, and (ii) undetected extreme transformations provide valuable information to the developers in order to improve their test suite.

Developers can improve a test suite by enhancing an existing test case with additional inputs, strengthening the assertions, or they can add a new test. Yet, the complex interactions between the test cases and the program under test make it challenging to decide which action to take. For example, a given method can be executed only at the end of a long sequence of invocations, or it can be private and indirectly executed by the test suite. In all these cases, developers need to spend significant effort to understand the complex chain of invocations between the test cases and the method, the effects of the method and how the extreme transformation affects the program state.

The fourth objective of this thesis is to generate concrete test improvement suggestions from undetected extreme transformations. In this chapter we present Reneri, a tool that automates the analysis of the test execution on both, the original and the transformed method. The tool also studies the interaction between the test suite and the method under test to assist the developer in the improvement of the test suite. Reneri implements a dynamic analysis inspired by the *Reachability, Infection, Propagation* model (*RIP* or *PIE*[1])[49, 24, 81]. Reneri takes as input a program, its test suite and the set of undetected extreme transformations. It then instruments the code of the program and the test suite to gather information about the program state during test executions. Reneri compares the information produced during the execution of the original code with the information from the execution of the extreme transformation.

Reneri determines the root cause of the undetected transformation: (i) the test inputs are not sufficient to *infect* the state of the program (**no-infection**); (ii) the infection does not *propagate* to the test cases (**no-propagation**); (iii) the test cases have a *weak oracle* that does not observe the infection (**weak-oracle**).

Reneri uses the diagnosis information to synthesize suggestions, in plain English, that can be directly followed by the developers to create a new test input, add a new assertion or create a new test case.

We perform three systematic evaluations of Reneri: (i) a quantitative analysis of root causes behind undetected extreme transformations; (ii) a qualitative analysis of the synthesized suggestions through interviews with developers; (iii) a quantitative evaluation of test generation tools to handle the undetected transformations.

The first evaluation is performed with 15 projects containing 312 undetected transformations. We observe that there is no dominant root cause to miss a transformation, there is balance between **no-infection**, **no-propagation** and **weak oracle**.

In the second evaluation, we use Reneri to generate improvement suggestions for a selected number of undetected extreme transformations in four open-source projects. Developers considered these suggestions helpful in most cases and also provided feedback on Reneri's strengths and weaknesses.

[1] Propagate Infect Execute

In the third evaluation, we design and implement two strategies to handle the undetected transformations: one based on EVOSUITE as a test generation alternative and another using DSpot as a test case improvement approach. In total, both strategies were able to generate test cases that detect 77% of the previously undetected extreme transformations.

The key contributions of this chapter are:

- Reneri, a new tool that automates the infection and propagation analysis on Java programs. The key novelty is that it generates natural language suggestions that can be followed by the developers to improve their test suite;
- an empirical assessment of the tool with 15 projects and a total of 312 undetected extreme transformations. This analysis indicates that undetected extreme transformations are mostly due to a lack of observation;
- a comparison between two strategies based on EVOSUITE and DSpot in the solution of undetected extreme transformations;
- open science: Reneri is available as open source [2] and empirical data is provided as open data [3] to facilitate future replication.

The rest of this chapter is organized as follows: first we describe extreme transformations through the class shown in Listing 1.1 and the test suite shown in Listing 1.2. We discuss how extreme transformations can be helpful to reveal weaknesses in the test suite; then we define the main concepts behind the dynamic analysis implemented in our tool and how we adapt the *RIP* model conditions to extreme transformations. In the following sections we describe the different stages of the proposed dynamic analysis and how they are implemented. Then, we expose the research questions we followed to validate our proposal, and the answers to these questions. Finally, we discuss the threats to the validity of our study and the conclusions that can be drawn from the results of this chapter.

4.1 Extreme transformations and test suite weaknesses

In this section we illustrate the possible interactions between a test suite, a program under test and extreme transformations. We show what types of weaknesses in the test suite these extreme transformations can reveal. We also illustrate how tailored dynamic analyses can be used to guide the developers for test suite improvement.

4.1.1 Examples of undetected extreme transformations

Recall Listing 1.1, it shows the `VersionedSet` class which implements a set that keeps track of the insertion operations through the `version` field. Listing 1.2 shows a JUnit test class that verifies `VersionedSet`. The test class contains three test methods implementing three test cases. These test cases are able to reach all but one method in the class under test, that is, at least one statement of each method is executed by a test case except for `getVersion`. All tests pass when executed with the original class code.

When applying Algorithm 2 (discussed in Section 3.1.1) to `VersionedSet`, using the transformations listed in Table 3.1 the following extreme transformations are applied: (1) remove the body of `add` (line 5) (2) remove the body of `incrementVersion` (line 13) (3) replace the body of `equals` by `return true` (line 33) (4) replace the body of `equals` by `return false` (5) replace the body of `intersect` by `return null` (line 46) (6) replace the body of `isEmpty` by `return true` (line 25) (7) replace the body of `isEmpty` by `return false`

Methods `size`, `contains` and `getVersion` are ignored as discussed in Section 2.1.2: `size` and `contains` delegate the execution to similar methods of the underlying instance of `ArrayList`. `getVersion` is a simple getter for the `version` field and it is not invoked by the test cases. Methods matching these patterns are automatically detected and skipped. This design decision relies on the observation that most developers do not target these types of methods directly in test cases.

[2] https://github.com/STAMP-project/reneri

[3] https://github.com/STAMP-project/descartes-amplification-experiments

The objective of the extreme transformations is to run the test class against each transformed method. This provides an assessment of the test suite ability to reveal these transformations. For our example, we observe the following: transformations (1), (4) and (5) are detected: at least one of the test cases fails when the transformation is performed. Meanwhile transformations (2), (3), (6) and (7) are not detected by the test suite.

4.1.2 Conditions to detect extreme transformations

In order for the test suite to detect an extreme transformation, the following conditions must hold (we adapt these conditions from the *Reachability, Infection, Propagation* model): (i) **reach:** at least one test case invokes the method modified by the extreme transformation (reaches the transformed method) (ii) **infect:** the program state after the method call must be different from the state observed when no transformation is performed (the transformation infects the state) (iii) **expose:** the modification propagates to a point in the top level code of a test case (the infection propagates to an observable point) (iv) **assert:** there is an oracle (*i.e.* an assertion) that verifies the modified state.

The **reach** condition depends on the coverage achieved by the test suite. As before, in this chapter we focus only on the analysis of methods that are reached by the test suite, that is, at least one statement in their code is executed.

For the **infect** condition, we consider the following: after the execution of the transformed method, we observe the state of the return value, the state of the object instance on which the method has been invoked, and the state of any of the actual parameters. We compare these values when running the test suite on the original and the transformed versions of the method. If any value is different between these executions, we consider that the interaction between the test suite and the extreme transformation can infect the program state.

An infection is **exposed** if it can be observed from a test case, *i.e.*, if the test case can query some program states that are affected by the infection at the boundary of the transformed method. The last condition about **assert** holds if the test case actually checks a property that is influenced by this visible state that is affected by the infection.

4.1.3 Analyzing undetected extreme transformations

The analysis of the previous conditions helps to explain why an extreme transformation is not detected by the test suite. We illustrate this with our example.

The `equals` method (line 33) is reached by `testEquals` (line 10). Yet, the extreme transformation that replaces the body of `equals` by `return true` has no effect on the immediate state of the program, when executed with `testEquals`. The `equals` method returns the same value for the given input and has no other side effect. In fact, the test case makes the method return `true` and no other test case makes the method return `false`. The transformation is reached in the execution but the program state is not infected by the extreme transformation. The effects are therefore not propagated and the transformation can not be detected. In this particular example it is necessary to augment the test suite with a new test case using a new input.

When transformation (2) is applied the method `incrementVersion` (line 13) and `testAdd` is executed, the immediate program state is infected: the value of `version` is 0 while it is 1 with the original method body. The infection is propagated to the code of the test case, as the state of the `list` variable is affected after the invocation of `add` in line 6. However, the test case is not verifying this state element. In this example it is necessary to add a new assertion targeting `getVersion`.

Both transformations (7) and (6) affect the method `isEmpty`. The former transformation makes the method return `false`. In the context of the `testIntersection` test case, this transformation is similar to (3) for the `equals` method. The program state is not infected. As for the latter transformation, the program state is infected. `isEmpty` should return `false` and not `true`. However, the return value of `intersect` turns out to be the same as expected in the original test case and therefore the infection is not propagated. This analysis could produce two possible outcomes. The utility of `isEmpty` for the implementation of `intersect` could be questioned. No matter the result of this method, the final effect is the same so the code could require some refactoring. On the other hand, a completely new test case could handle these two transformations. In particular, a test case that can target directly the `isEmpty` method, as it is public.

According to Definition 3, of the three methods mentioned before, only `incrementVersion` is pseudo-tested. `equals` and `isEmpty` are both required according to Definition 4, yet they are related to undetected extreme transformations. This indicates that, beyond pseudo-tested methods, studied in Chapter 3, undetected transformations in required methods may signal testing issues.

The three examples above also illustrate how we can obtain valuable insights when analyzing the interactions between test cases and extreme transformations at the immediate program state after the transformation. This analysis can help developers to decide whether they should add assertions to an existing test case, or add a new test case or if the program is not fully testable and requires refactoring.

4.2 Automatic Synthesis of Suggestions for Test Improvement

The automatic synthesis of suggestions for test improvements takes as input a Java program, its test suite, and a set of undetected extreme transformations. The approach consists in automatically understanding why a given extreme transformation is not detected and in suggesting the developers a possible test improvement to detect it. These understanding and suggestion synthesis activities are based on the comparison of the test suite execution on the original and the transformed program.

In this section, we provide precise definitions of the program analysis concepts that we manipulate to understand the effect of extreme transformations. Then, we detail each step of the dynamic analysis and test improvement suggestion synthesis.

4.2.1 Definitions

We now rigorously define our terminology for state observation. Our definitions are scoped by the Java language and the JVM. We believe they could be extrapolated to other languages and runtimes in general.

Definition 5. ***Basic state of a value:*** *Let v be a Java typed value, the basic state of v is a set of property-value pairs BS defined as follows:*

- *If the type of v is any Java primitive type, any wrapper type or `String` , then $BS = (value, v)$.*
- *If v is a reference to an object, then BS contains $(null, b)$ where b is a boolean value indicating whether v is the `null` reference.*
- *If v is a reference to an array, then BS contains $(length, l)$ where l is the length of the array.*
- *If the type of v implements interfaces `java.util.Collection` or `java.util.Map`, then BS contains $(size, s)$ where s is the size of the collection.*

Definition 6. ***State of value:*** *The state of a given value v is a set of properly-value pairs S defined as follows:*

- *S contains all the elements in the basic state of v.*
- *If the type of v is not a Java primitive type, a wrapper type or `String` then S contains the basic state of all values of all fields, declared or inherited by the type of v.*

For example, the state of a reference to an instance of the `VersionedSet` class in Listing 1.1 right after its creation would be: $\{(null, false)\ (version, 0), (elements.null, false), (elements.size, 0)\}$.

As in previous chapters, here we consider a program P to be a set of method and its accompanying test suite T to be a set of test cases.

Definition 7. ***Extreme transformation:*** *We consider an extreme transformation to be a tuple (m, m') where $m \in P$, m' is the transformed variant of m, also called transformed method. m' is obtained from m by removing all the instructions in its body and adding a single* ***return*** *instruction with a predefined value if m returns a value. The predefined values according to the return type of m are shown in Table 3.1. An extreme transformation is said to be* undetected *if $\nexists t \in T$ that fails when m is replaced by m' in P and the tests in T are executed.*

Our analysis takes as input a program P, a test suite T and U, a set of undetected extreme transformations as defined in Definition 7.

Definition 8. ***Local immediate program state after a method invocation:*** *The local immediate state of a program P after the execution of a method $m \in P$ is a set of property-value pairs MS_m defined as follows:*

- *MS_m contains all elements in the state (as defined in Definition 6) of all the arguments of m, if any.*
- *If m has a return value, (i.e. m is not `void`), then MS_m contains all elements in the state of the result value as defined in Definition 6.*
- *If m is not static, then MS_m contains all elements in the state (as defined in Definition 6) of the instance on which m was invoked.*

$MS_{m,t}$ denotes the immediate program state after the invocation of m when executing the test case $t \in T$.

As an example, if a program creates an instance of `VersionedSet` and then invokes `addElement` on this instance with a `String "something"`, the local program state would be: $\{(null, false)$ $(version, 1)$, $(elements.null, false)$, $(elements.size, 1)$, $(value, "something")\}$. Notice how this state includes the state of the $VersionedSet$ instance and the state of the argument. In this case the method is `void`, so the result value is not reflected in the state.

Definition 9. ***Local immediate program state infection:*** *Let $m \in P$ be a method that is reached (executed) by at least one test case $t \in T$. Let $(m, m') \in U$ be an undetected transformation. We say that the execution of the test case t provokes an infection of the local immediate program state under the extreme transformation if $MS_{m,t} \neq MS_{m',t}$. That is, there is an infection if we are able to observe a difference in the immediate program state after the execution of m by t and the execution of m' by the same test case t.*

Definition 10. ***Test case state:*** *Let $t \in T$ be a test case. Let O be the set of values local to t, that is, all the values that result from any expression or subexpression in the code of t. Then, TS_t denotes the state of t and it is the union of the states of all values local to t. That is $\cup_{o \in O} S_o$. $TS_{t,m}$ denotes the state of t when the original method $m \in P$ is used and $TS_{t,m'}$ denotes the state of t when it is executed with m'. TS_m denotes the union of the state of all tests in the test suite when executing the original method, that is, $TS_m = \cup_{t \in T} TS_{t,m}$. Analogously, $TS_{m'}$ represents the union of the state of all tests executed under the extreme transformation.*

For example, the state of `testEquals` in line 10 of Listing 1.2 includes the state of the set instances referenced by `one` and `two` and the result of the result of the invocation to the `equals` method.

Definition 11. ***Program state infection propagation:*** *Let $m \in P$ be a method and $(m, m') \in U$ an undetected extreme transformation. The immediate program infection is said to propagate to the test case $t \in T$ if there is a difference between the state of t while executing m and the state of t when executing m'. That is if $TS_{t,m} \neq TS_{t,m'}$.*

As per the definitions above, we study the local state of a program after a method invocation and the local state of a test case. The former includes the state of the receiver of the method, the state of the arguments, and the state of the resulting value. The latter includes the state of all values resulting from an expression in the code of the test case.

4.2.2 Overview of the process for test improvement suggestions synthesis

The global process to understand undetected extreme transformations and to synthesize a test improvement suggestion operates in three main stages. Algorithm 3 outlines the process. Each stage is detailed in the following sections:

1. Infection detection: identify the extreme transformations that infect the local immediate state; (line 1, Section 4.2.3)

2. Propagation detection: discover which infections reach some test case states; (line 2, Section 4.2.4)

3. Test improvement suggestion: generate the report by consolidating the information gathered in the two previous stages (line 3, Section 4.2.5).

The initial two stages include a dynamic analysis of the program. Each stage instruments the elements to be observed and executes the test suite with the original code and the transformed method variant. These executions of the instrumented program record the program and test states to help the suggestion synthesis. The states are compared to discover the symptom that explains why each transformation was not detected by the test suite.

```
Data: P: program under test, T: test suite, U: undetected extreme transformations
1 no-infection, infection ← find_infections(P, T, U)
2 no-propagation, weak-oracle ← find_propagations(P, T, infection)
3 gen_suggestions(no-infection, no-propagation, weak-oracle)
```

Algorithm 3: The three stages process implemented by Reneri

The analysis results in the identification of one symptom for each undetected extreme transformation. The identified symptom shall help to explain why the extreme transformation was not detected by the test suite. We identify the following three types of symptom:

- **no-infection** (**ni**): there is no observable difference in the local state of the program after the method invocation when the test case is executed on the original and transformed methods.

- **no-propagation**(**np**): there is an observable difference in the program state after the method invocation, but there is no observable difference in the state of the test case.

- **weak-oracle** (**wo**): the program state infection is propagated to the code of a test case but no assertion fails.

These symptoms, as well as the intermediate observations of the dynamic program analysis are consolidated in a final report to provide concrete improvement suggestions to the developers.

4.2.3 Infection detection

```
Data: P: program under test, T: test suite, U: set of undetected extreme transformations
Result: no-infection: {no-infection symptoms}, infection: {extreme transformations that
        produce an infection}
1  function find_infections(P, T, U) does
2    foreach (m, m') ∈ U do
3      T_m ← { t ∈ T : t executes m }
4      P_i ← instrument m in P
5      MS_m ← observe(P_i, T_m)
6      P' ← replace m by m' in P
7      P'_i ← instrument m' in P'
8      MS_m' ← observe(P'_i, T_m)
9      diff ← get_diff(MS_m, MS_m')
10     if diff ≠ ∅ then
11       infection ← infection ∪ {((m, m'), diff)}
12     else
13       no-infection ← no-infection ∪{(m, m')}
14     end
15   end
16   return no-infection, infection
17 end
```

Algorithm 4: Infection detection stage of the process. This stage identifies the extreme transformations that are not detected due to a **no-infection** symptom

```
1  function observe(P, T) does
2      for i ← 1 to N do
3          S_i ← execute T with P
4      end
5      return ∪_{i=1}^{N} S_i
6  end

7  function get_diff(S_1, S_2) does
8      diff ← ∅
9      foreach (p, v) ∈ S_1 do
10         if ∃(p, w) ∈ S_2 ∧ v ≠ w then
11             diff ← diff ∪ {(p, v, w)}
12         end
13     end
14     return diff
15 end
```

Algorithm 5: Two functions to compute the state invariants across test executions and find a difference between the state of the original program and the transformed variant.

The goal of the first stage is to determine whether the execution of the tests under an extreme transformation infects the program state. We instrument the original program and the program after the application of the extreme transformation to observe the immediate program state in both situations. If we can observe a difference between the states, then this signals an infection. The main steps of this stage are outlined in Algorithm 4

Instrumentation

For each method and its transformed variants we observe the following parts of the program state: the instance on which the method is invoked, all arguments (if any), and the result value (if the method returns a value).

For primitive values and strings, the observation is trivial. For objects of other types, an observation means inspecting public and private fields using reflection. In this way the observation minimizes potential side-effects to the object being observed.

In practice, we obtain these observations by instrumenting the code of the original methods and its transformed variant (lines 4 and 7 from Algorithm 4). The instrumentation targets the compiled bytecode of the program.

Listing 4.1 shows an example of how the `equals` method from Listing 1.1 is instrumented for observation. The `observe` function saves the state of the object as defined in Definition 6.

```
1 public boolean equals(Object other) {
2  boolean result = ... /* original method code */
3  /* Observation */
4  observe(this);
5  observe(other);
6  observe(result);
7  return result;
8 }
```

Listing 4.1: Method instrumented to observe the immediate program state

Dynamic Analysis

Each extreme transformation $(m, m') \in U$ is observed and analyzed independently. After the instrumentation, all the test cases known to cover m are executed. With this, we observe the original program state MS_m. The same is done with m' to obtain $MS_{m'}$. If there is at least a property with a different value in both states, then the states are said to be different (`get_diff` in Algorithm 5). The property points to the code location where the infection has manifested: one of the arguments, the receiver or the result of an invocation.

If no difference is observed, then the process has discovered a **no-infection** symptom for the given extreme transformation.

A method could be invoked several times by the same test case. In our experience, we have seen methods invoked thousands of times in a single test execution. During the dynamic analysis of this stage, each method invocation is uniquely identified by its order in the entire method invocation sequence. This identification differentiates the states from each invocation of m and m'. The properties from one method invocation are distinguished from the properties of another method invocation. For example, the result value of the first invocation of m is considered as a distinct property from the result value of the second invocation of m. With this, the state elements from the first invocation of m are only compared to the same state elements form the first invocation of m' and so on.

During the observations, there may be changes from one execution to the other, that are not due to the extreme transformations [14]. Such variations may be derived from random number generation, usage of system time, usage of temporary file paths, and they should be discarded. As a trivial example, in line 7 of Listing 4.2, the value of `start` depends on the current system time. In order to identify such changing values, we execute the tests N times when recording the states of the original methods and another N times when recorded their transformed variants. N is a parameter that can be controlled by the user with a default value of 10.

```java
@Test
public void handlesManyChildren() {
 StringBuilder longBody = new StringBuilder(500000);
 for (int i = 0; i < 25000; i++) {
  longBody.append(i).append("<br>");
 }
 long start = currentTimeMillis();
 Document doc = Parser.parseBodyFragment(longBody);
 assertEquals(50000, doc.body().childNodeSize());
 assertTrue(currentTimeMillis() - start < 1000);
}
```

Listing 4.2: Example of a test case using the system time. The value of `start` changes on every tests excution

For the state comparison, we keep only the properties that had the same value across all N executions. So, we actually consider as the state of the program only the set of state elements that remained unchanged across executions, as can be seen in the `get_diff` function from Algorithm 5.

If a difference is observed, we can collect the state property, that is, whether the infection can be observed in a field or the value of the result of the method, in one of the arguments or the receiver.

After this stage, the undetected transformations can be partitioned in two groups: those exhibiting a **no-infection** symptom, and those for which the test executions produce the infection of the immediate program state.

4.2.4 Propagation detection

The goal of this second stage is to determine which program state infections, detected in the previous stage, are propagated to the test code. In this stage we observe the state of each test case covering these transformations. At the end, we are able to detect whether the infection could be observed in the existing test code (**weak-oracle**) or not (**no-propagation**). The main steps of this stage are outlined in Algorithm 6.

Instrumentation

In this stage, the test states are also observed through source code instrumentation. Unlike in the previous stage, here we instrument the Java source code of the test cases. This instrumentation observes the test case state as defined in Definition 10. We instrument only the tests that cover the method of at least one extreme transformation for which a program infection was previously detected. The code is instrumented to observe the state of the values produced by all expressions

```
    Data: P, T, infection
    Result: no-propagation, weak-oracle
 1  function find_propagations(P, T, infection) does
 2      foreach (m, m', diff_m) ∈ infection do
 3          T_m ← { t ∈ T : t executes m }
 4          T_{m,i} ← instrument T_m
 5          TS_m ← observe(P, T_{m,i})
 6          P' ← replace m by m' in P
 7          TS_{m'} ← observe(P', T_{m,i})
 8          diff_t ← get_diff(TS_m, TS_{m'})
 9          if diff_t ≠ ∅ then
10              weak-oracle ← weak-oracle ∪{((m, m'), diff_t)}
11          else
12              no-propagation ← no-propagation ∪{((m, m'), diff_m)}
13          end
14      end
15      return no-propagation, weak-oracle
16  end
```

Algorithm 6: Second stage of the process. This stage identifies **no-propagation** and **weak-oracle** symptoms

and subexpressions in the code of the test case. This instrumentation amplifies the observation capabilities of the test cases, limited before only to the values being asserted by the developers.

Listing 4.3 shows the instrumentation applied to the `testAdd` test method from Listing 1.2.

We exclude from the observation: constant expressions and expressions on the left side of an assignment. Generic test method and test classes, that is with type arguments, are not considered in our current implementation. The instrumentation inserts `try...catch` blocks in the test code, in order to observe exceptions that could be thrown during test execution, as it is done in line 13.

```java
@Test
public void testAdd() throws Exception {
 try {
  VersionedSet list =
   observe(new VersionedSet());
   observe(list).add(1); // State of list
   assertEquals(1,
    observe( // Result of size
     observe(list).size() // State of list
    ));
 }
 catch(Exception exc) {
  observe(exc);
  throw exc;
 }
}
```

Listing 4.3: Test case instrumented to observe infection propagation

Dynamic analysis

The dynamic analysis in this stage is very similar to the infection detection. Each extreme transformation is analyzed in isolation. The test are executed N times for the original code and N times for the transformed code. Only the state elements that remained unchanged are kept for the state comparison. Unlike the previous stage, here the states are recorded from the test code and the properties point to code locations where developers can actually insert new assertions. If no difference was observed between the execution of the original method and the transformed method, then a **no-propagation** symptom is signaled. The previously detected infection did not propagate to an observable point in the test code. On the contrary, if a difference is observed, then we signal a **weak-oracle** symptom. The existing assertions do not notice the extreme transformation.

The body of the method `example.VersionedSet.equals(java.lang.Object)` was replaced by
`return true;`
yet, `example.VersionedSetTest.testEquals` did not fail.

When the transformed method is executed, there is no difference with the execution using the original source code.

This could mean that the original method always returns the same value. Consider creating a modified variant of the test mentioned above to make the method produce a different value.

Figure 4.1: Suggestion synthesized for a **no-infection** symptom related to the `equals` method in our example.

4.2.5 Suggestion synthesis

The third stage synthesizes the final suggestions. The suggestions are compiled into a human readable report. The report includes a detailed diagnosis of the symptom for each extreme transformation and an indication of the potential solution strategies. In this section we discuss how these reports are generated, the exact information they include and provide an example for each symptom.

Suggestion for a no-infection symptom

For a given undetected extreme transformation, if no immediate program infection is observed for all tests case, it means that the test input is not able to generate a state difference (**no-infection**). It also means that the method had no observable side-effects over the instance on which it was invoked or the arguments. The result value of the method is the same across all invocations. However, there are test cases that reach the method.

Altering the existing input in these test cases could alter the program state so the method can return a different value or produce a different side effect. So, the suggestion generated in the process tells developers *to create new test cases from the ones covering the method* and alter their input to induce an observable effect.

Consider again the `equals` method in Listing 1.1. The suggestion given to the developer is to create a new test case to make the method return a different value using `testEquals` as a starting point. Figure 4.1 shows the report.

Suggestion for a no-propagation symptom

A **no-propagation** symptom unveils an immediate program infection that is not propagated to the test code. A different program state is observed after one or more invocations of the transformed method. However, no state difference is observed from the test cases. The effects of the program infection are masked at some point in the execution path. Observing the changed state requires a new invocation sequence able to propagate the infection to an observable point from the test. This means that *new test cases are then required to detect these extreme transformations*. This is the suggestion we generate for these symptoms.

However, the method under study could be private and not directly accessible from the test cases. To address this problem, we find methods that are accessible from the test code, and that are as close as possible in an invocation sequence to the method we want to target. For this we perform a static analysis in the code of the project that follows the invocation graph. The analysis produces the list of public or protected methods inside accessible classes that can be use to reach the method.

The report for a **no-propagation** symptom includes a description of the transformation and the list of methods that should be targeted in the new test cases. If the method is already accessible then it is the only one listed as target. If it is not accessible, we include the list produced by the static analysis. The report also includes the test cases executing the method.

The body of the method `example.VersionedSet.isEmpty()` was replaced by `return true;` yet, `example.VersionedSetTest.testIntersection` did not fail.

It is possible to observe a difference between the program state when the transformed method is executed and the program state when the original method is executed. This difference is observed right after the method invocation but not from from the top level code of any test.

For one invocation of `isEmpty`, it was observed that the return value was `true` but should have been `false`.

To solve this problem you may consider to:

- Create a new test case that targets the result of `isEmpty` directly, since it could be accessed from a test class.
- Refactor the code that uses this method. Maybe the method is not actually needed in the context that it is being used.

Figure 4.2: Suggestion synthesized for a **no-propagation** symptom related to the `isEmpty` method in our example.

In our example, the `isEmpty` method has a **no-propagation** symptom. As, it is accessible from the test code, the suggestion is to create a new test case targeting `isEmpty` directly. Figure 4.2 shows the report.

Suggestion for a weak-oracle symptom

If an infection propagation is detected (**weak-oracle**), it is visible in the result of an expression in the test code. In the previous stage we have recorded the exact code location of this expression.

The **weak-oracle** suggestion is *to add an assertion at the right location in the test code*. We even provide the developer with the value to be asserted.

The state difference could be the result of the expression itself if it is a primitive type or a string. But, if the expression returns an object, the difference may be observed in one of its fields. For instance, in our example from Listing 1.2 in line 21 the result of the expression at the right of the assignment is a `VersionedSet`. With the original code, the value of the `version` field should be `1` while, with the transformed method it is observed to be `0`.

If the state difference is observed through an accessible field of the result of the expression or the result itself, the suggestion is to create an assertion targeting this value.

If the field is not accessible, (*i.e.* it is private) further actions are required. For this, we perform a static analysis to find methods that use the identified field. These methods could be not accessible from the test code. So, we find accessible class members invoking these methods, in the same way it is done for the **no-propagation** reports. The final suggestion is to assert the result and side effects of the final set of methods, invoked in the result of the initially identified expression.

In Listing 1.1, the process suggests the addition of a new assertion targeting the `getVersion` method invoked in the list instance of line 7. Figure 4.3 shows the synthesized suggestion.

4.2.6 Implementation

We have implemented our proposal in an open-source tool which we named Reneri. The tool has been conceived as a Maven plugin and it is able to target Java programs that use Maven as main build system. The code and all data related to this chapter are available from Github [4] [5]. Reneri relies on Javassist 3.24.1 [11] for bytecode instrumentation, Spoon 7.1.0 [58] for static code analysis and Java source code transformation and Descartes 1.2.5 [80] to apply extreme

[4] https://github.com/STAMP-project/reneri

[5] https://github.com/STAMP-project/descartes-amplification-experiments

The body of the method `example.VersionedSet.incrementVersion()` was removed yet, none of the following tests failed:

- `example.VersionedSetTest.testIntersection`
- `example.VersionedSetTest.testAdd`

It is possible to observe a difference between the program state when the transformed method is executed and the program state when the original method is executed. This difference can be observed in `VersionedSetTest.java` from the expression returning a value of type `example.VersionedSet` located in line 22 from column 31 to column 34

When the transformation is applied to the method, it was observed that the field `version` of the value obtained from the expression was `0` but should have been `1`.

Consider modifying the test to verify the value of `version` in the result of the expression.

Consider verifying the result or side effects of one of the following methods invoked for the result of the expression:

- `example.VersionedSet.getVersion()`

Figure 4.3: Suggestion synthesized for a **weak-oracle** symptom related to the `incrementVersion` method in our example.

transformations. All stages in the process described before are exposed as Maven goals. Apart from human readable reports, Reneri also generate files more suitable for automatic analysis that could be used by external tools.

4.3 Experimental Evaluation

We assess our proposal based on a set of research questions. In this section we present these questions, the projects we used as study subjects and the results we obtained.

4.3.1 Research questions

RQ1 *To what extent does the execution of an extreme transformation infect the immediate program state?*

With this question we quantify (i) how frequent the **no-infection** symptom is and (ii) how often an infection can be observed. We collect the number of extreme transformations exhibiting an infection across projects.

RQ2 *To what extent can test cases propagate the effects of extreme transformations to the top level test code?*

With this question we quantify how frequently **no-propagation** and **weak-oracle** symptoms appear across projects. **weak-oracle** symptoms might represent assertions missed by developers. **no-propagation** symptoms may require the creation of new test cases.

RQ3 *Are the suggestions synthesized by Reneri valuable for the developers?*

In this question we empirically assess the suggestions generated by Reneri. The goal is to know if explaining the undetected extreme transformations can actually help developers to improve their test cases.

RQ4 *Can developers leverage test improvement tools to deal with undetected extreme transformations?*

With this question we explore if developers can leverage state-of-the-art automatic test generation and test improvement tools. We also quantify the effectiveness of these tools against **no-infection**, **no-propagation** and **weak-oracle** symptoms.

Table 4.1: Projects used as study subjects. The first column is the name of the project. *App LOC* and *Test LOC* show the lines of code for the application code and the test code respectively. *Tests* Shows the number of test cases as reported by Maven. *Methods* shows the number of application methods with at least one undetected extreme transformation. *Transformations* shows the number of undetected extreme transformations in the project. *Min #Test*, *Max #Test* and *Avg #Test* refer to the minimum, maximum and average number of test cases in *Test* that execute each method in *Method*

Project	App LOC	Test LOC	Tests	Methods	Transformations	Min #Test	Max #Test	Avg #Test
`jpush-api-java-client`	3667	3112	72	2	2	1	5	3
`commons-cli`	2800	4287	471	3	5	8	154	61
`jopt-simple`	2410	6940	838	3	7	1	77	26
`yahoofinance-api`	2925	453	15	3	5	3	4	4
`gson-fire`	1665	1644	79	4	4	2	17	7
`j2html`	3460	1250	51	5	5	1	1	1
`spring-petclinic`	731	687	40	6	6	2	11	5
`javapoet`	3363	5334	340	10	10	1	104	12
`eaxy`	3599	1670	204	11	14	1	63	18
`java-html-sanitizer`	7662	5944	260	13	14	1	134	44
`cron-utils`	4544	5202	466	16	16	1	207	49
`commons-codec`	8241	11957	889	20	22	1	96	21
`jsoup`	12015	7911	680	33	38	1	483	50
`TridentSDK`	5544	1670	127	35	46	1	17	3
`jcodemodel`	13752	1544	68	95	118	1	50	11
Global	76378	59605	4600	259	312	1	483	20

4.3.2 Study subjects

We select a set of 15 open-source projects as study subjects to answer our research questions. All projects use Maven as the build system, are written in Java (8 or lower) and use JUnit(4.12 or lower). We selected these projects systematically as follows. We started from 51 single-module projects, matching the above conditions and studied by the authors of [76] and included in Chapter 3. We focus on single-module projects as they have a very well defined structure and all unit test cases are well located. This eliminates the need to deal with the idiosyncrasies from specific projects.

For these 51 initial projects, we performed the following actions: (i) Clone the repository (ii) Switch to the commit of the last stable release (according to the Github API) or the latest commit if the API produced no result (iii) Build the project using `mvn clean test` (iv) Execute Descartes to find undetected extreme transformations.

The last step computes the set of undetected extreme transformations and the test cases reaching these transformations. The full workflow was successful in only 25 projects. For the other 26 projects the main reasons of failure were compilation errors, failing tests or specific set-up requirements or dependencies external to the Maven ecosystem.

From these 25 projects, we kept those having at least one undetected extreme transformation. We discarded projects with more than 100 application methods with at least one undetected extreme transformation. With this we eliminated 6 projects where all extreme transformations were detected and 2 that had more than 100 affected methods. 2 additional projects were discarded due to technical incompatibilities with Reneri: one of them widely uses generic classes in the test code, which we do not target in our current implementation. The testing configuration of the other project clashed with the way we log the program state information during test execution.

Table 4.1 shows the list of projects used in the present study. The columns in the table show the number of lines of the main source code (LOCs) as measured by *cloc*[6], the number of lines for the test code, the number of test cases in the project as reported by Maven with the execution of `mvn test`, the number of application methods for which at least one extreme transformation was not detected and the number of undetected extreme transformations for the entire project.

The last three columns show the minimum, maximum and average number of test cases executing each method included in the *Methods* column.

Appendix B shows the code revision used for each project.

All projects included in the study are of small to medium size (731 to 13752 LOCs). In seven cases, the size of the test suite is comparable or larger to the size of the application code. The

[6]https://github.com/AlDanial/cloc

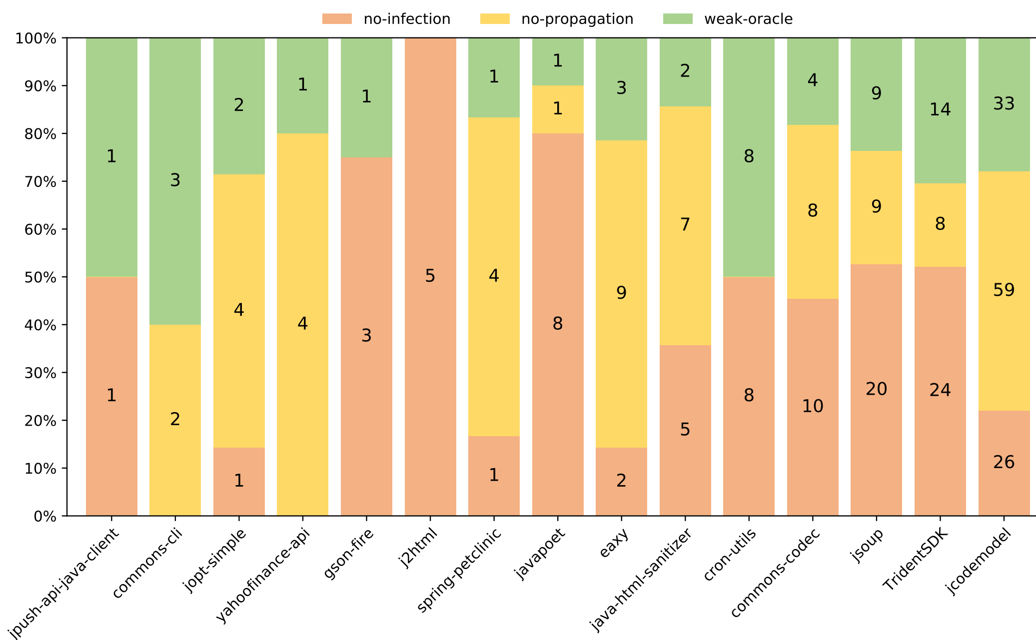


Figure 4.4: Number and proportion of **no-infection** in orange, **no-propagation** in yellow and **weak-oracle** in green found for each project.

size is not always correlated to the number of test cases which range from 15 to 889. There are 260 methods with at least one undetected extreme transformation, ranging from 2 to 95 per project. The total number of transformations studied is 312, distributed from 2 to 118 per project. In most projects there is at least one transformation that is covered by only one test case. In four projects all the transformations are executed by more than one test case and in `commons-cli` the methods are covered by at least eight test cases. Notably, in five projects, one of the identified methods is covered by more than 100 test case: in `jsoup`, one transformation is covered by 483 test cases. The transformations are covered on average by 20 test cases.

4.3.3 RQ1: *To what extent does the execution of an extreme transformation infect the immediate program state?*

To answer **RQ1** we execute the first stage of Reneri and collect the number of **no-infection** symptoms for all our 15 study subjects.

We observe an immediate program infection for 198 out of the 312 undetected extreme transformations. That is, in 63% of the cases, the existing test inputs do trigger an infection of the local program state at the transformation point. The rest 114 (37%) does not provoke a program infection. So, the execution of the majority of transformations provokes a program infection.

Figure 4.4 shows the number and proportion of the three symptoms: **no-infection**, **no-propagation** and **weak-oracle** found for each study subject. For instance, `jpush-api-java-client` has 50% of cases with **no-infection** and 50% of cases with **weak-oracle**. In all but three projects the **no-infection** symptoms are less than 52% of all symptoms discovered. In two projects, `commons-cli` and `yahoofinance-api` the execution of all extreme transformations infected the program state. On the contrary, in `j2html` all the symptoms are `no-infection`. So, we observe that the proportion of symptoms is different from one project to another.

A **no-infection** symptom occurs when the effects of the transformed method do not differ from the effects of the original method, with respect to the existing test input. As an example, one of the methods in `j2html` is an `equals` method executed by only one test case where it returns `true`. There is no test case where the method should return false, that is, it is never presented with objects that are not equal.

The other four methods in the same project are all void and are intended to perform a sanitization in the fields of the class instance, if needed. These four methods are executed by only one test case whose input does not contain a string requiring the sanitization. Thus, all these methods

are executed and the expected program state is the same under the extreme transformations. So, here, even when the test code executes these methods, the input does not trigger a behavior where their effects can be manifested.

One common cause that prevents an immediate program infection is related to boolean methods. In many cases these methods are tested to return only one value: always true or always false. In such a situation, the transformed method will have the same result in all invocations as the original method. This produces a **no-infection** symptom. In fact, in our study, 129 undetected extreme transformations were discovered in boolean methods. 65 of them do not provoke a program infection. Notably, 19 were created in `equals` methods and none of them induced a program infection.

So, boolean methods are related to a large portion of undetected extreme transformations (41%) and half of them exhibit a **no-infection** symptom.

Another common scenario comes from methods checking preconditions that are not tested with corner cases. Listing 4.4 shows one of those methods. As can be seen, these are void methods that check the value of a boolean expression (line 2), which is the precondition. If the expression is false, then an exception is thrown as in line 4. So, apart from the exception, these methods have no other effect. It is common that developers do not create test cases with inputs that do not meet the precondition.

```
1 private void checkMaxOneMatch() {
2  if (size() <= 1) return;
3  String message = ...
4  throw new IllegalArgumentException(message);
5 }
```

Listing 4.4: A **no-infection** example. The precondition is always satisfied in the test cases. the corner case is not verified

Where are the infections detected?

Reneri observes program infections in the result of the invocation of the transformed method, its arguments after the invocation and the instance on which the method was invoked. In this section we explore how frequently the infection is observed at each location.

The result value can only be observed in non-void methods. In this study we analyze 312 extreme transformations: 254 of them were created in non-void methods. 176/254 transformations in non-void methods are related to an infection and 167/176 (95%) of these infections are observed in the state of the result value. So, in non-void methods, the infection is observed in the result value in a vast majority of cases.

An infection in the arguments can only be observed for parameterized methods. Our study includes 227 transformations in methods with parameters. 144/227 of them caused an infection. We observe that 37/144 (26%) of these infections are detected in the state of the arguments. That corresponds to the best practice that the returned value stores the effect, not the argument.

An effect in the receiver instance can only be observed in non-static methods: 251 transformations in our study were created in instance methods. 160/251 of them provoked an infection and 69/160 (43%) can be detected in the state of the receiver instance.

In global terms, of the 198 transformations that caused an infection, 84% (167) can be observed in the state of the result value of the method, 19% (37) can be observed in the state of the arguments and 35% (69) can be detected in the state of the receiver instance. While, these numbers are more a reflection of the coding practices than of extreme transformations and even testing, they still provide valuable insights for developers who wish to understand undetected transformations.

Table 4.2 shows the details for the study subjects. The columns are divided in three groups. The first group corresponds to the transformations created in non-void methods, how many of them caused an infection and how many of them were detected in the state of the result value. The two other groups show analogous numbers for transformations in methods with parameters, non-static methods and whether the transformations can be detected in the state of the arguments and the receiver instance. The numbers for each project do not differ much from the general view. Only `spring-petclinic` shows more infections in the state of the arguments (4) less in the receiver instance (2) and none in the state of the result value.

Table 4.2: Number of times extreme transformations were detected in the state of the result value, the arguments or the receiver instance. The table is divided in three groups: non-void methods, parameterized methods and non-static methods. Each group shows the total number of transformations, how many of them provoke an infection and if they were detected in the result, the arguments or the receiver respectively.

	Transformations in non-void methods			Transformations in parameterized methods			Transformations in non-static methods		
Project	Total	Infection	In result value	Total	Infection	In parameters	Total	Infection	In receiver instance
jpush-api-java-client	2	1	1	1	0	0	1	0	0
commons-cli	5	5	5	4	4	0	2	2	0
jopt-simple	7	6	6	7	6	0	6	6	0
yahoofinance-api	5	5	5	5	5	0	0	0	0
gson-fire	4	1	1	4	1	1	3	1	1
j2html	1	0	0	1	0	0	5	0	0
spring-petclinic	2	1	0	6	5	4	6	5	2
javapoet	9	2	2	9	1	1	8	1	1
eaxy	12	11	10	7	5	0	14	12	5
java-html-sanitizer	9	7	5	10	5	2	7	6	2
cron-utils	14	8	8	16	8	0	14	7	2
commons-codec	22	12	12	20	10	2	17	8	2
jsoup	27	15	15	24	13	3	33	16	6
TridentSDK	44	22	21	33	17	2	40	20	6
jcodemodel	91	80	76	80	64	22	95	76	42
Total	254	176	167	227	144	37	251	160	69

Answer to RQ1

The existing tests infect the immediate program state at the transformation point in 63% of the cases. This indicates that undetected extreme transformations are mostly due to a lack of observation in the test cases. 84% of transformations infecting the program state result in a different return value for the transformed method.

4.3.4 RQ2: *To what extent can test cases propagate the effects of extreme transformations to the top level test code?*

To answer this question we execute the second stage of our process and collect all **no-propagation** and **weak-oracle** symptoms. We observe an infection that successfully propagates (i.e., a **weak-oracle**) for 83 extreme transformations, which represents 42% of the 198 transformations causing a program infection (and 27% of all transformations in the study). The other 115 transformations that can infect the program are diagnosed as **no-propagation**, which represents 58% of the transformations that do infect the program state (37% of all transformations). A general observation is that a weak oracle is clearly not the main symptom that prevents the test suite from detecting the extreme transformations.

Here again we notice differences among projects. In Figure 4.4 we notice four projects that do not have a single case of **no-propagation**. In three of them, all program infections reached the test code at locations that can be verified. The other, `j2html` had no program infection, which was discussed previously. In 11 projects, the **weak-oracle** symptoms are close or below 30% of all symptoms detected. In `commons-cli`, `jpush-api-java-client` and `cron-utils` these symptoms were majority or reached the 50% of all symptoms discovered for the project.

We hypothesize that the distance to the test code might be a factor that influences **no-propagation** symptoms. If the methods are far from the test code in the invocation sequence, then their effects are more likely to get hidden or lost before reaching the test code. To check this hypothesis, we compute the stack distance from the method to the test case for all transformations. We consider the stack distance to be the number of activation records in the method call stack between the activation record of the test case and the activation record of the method related to the transformation. If the method is invoked more than once in the test suite, we keep the lowest stack distance.

Figure 4.5 shows the distributions of these distance measures between the test and the transformed method. Yellow areas correspond to **no-propagation** symptoms and green areas correspond to **weak-oracle** symptoms. The higher the values, the larger the stack distance is. The

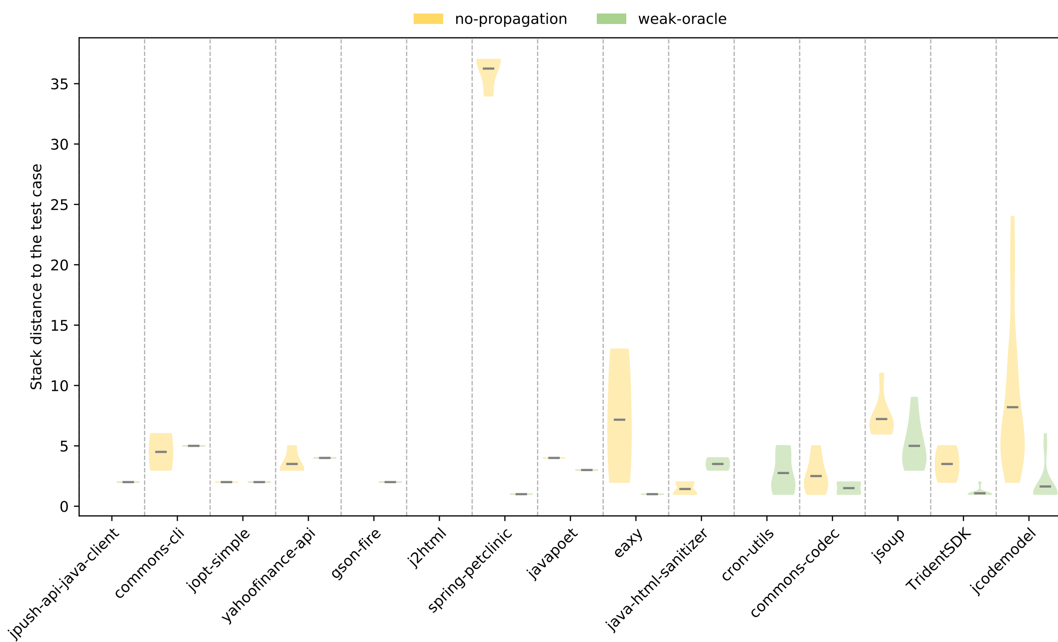


Figure 4.5: Distributions of the stack distance from the methods to the test cases for **no-propagation** symptoms (yellow) and **weak-oracle** symptoms (green).

figure also shows the mean value for each distribution. For example, in `commons-cli` the stack distances for **no-propagation** symptoms range from three to six with a mean value of 4.5 while all the **weak-oracle** symptoms had a distance of five activation records.

We observe that, in most cases, the mean stack distance of **weak-oracles** tends to be lower than the mean distance for **no-propagation** symptoms. This evidence confirms our hypothesis: transformations that occur far from the tests tend to propagate less than the others. Smaller projects tend to behave differently. The few **weak-oracle** cases that have higher stack distances occur on small projects, e.g., `java-html-sanitizer` or `jopt-simple` where they have the same distance. Larger projects,`jsoup`, `TridentSDK`, `jcodemodel` show a more significant tendency to the stack distance difference between these two symptoms.

no-propagation examples

An infection is not propagated when the code executed between the method invocation and the top level code of the test case *masks* the state infection. In many cases this happens for methods that are reached through a long sequence of invocations. An example from `commons-cli` is given in Listing 4.5: when the body of `isLongOption` (line 2) is replaced by `return false`, the test suite does not fail. This method is reached through a sequence of five invocations starting from `parse` (line 16). This sequence of invocations is triggered only when specific conditions are met (line 12). `isLongOption` is invoked 32 times by seven test cases in the test suite. The method is expected to return true in only one invocation. When the method is transformed to return only false, the infection is observed for the invocation in which it should have returned true. However, in this case, the `isShortOption` (line 5) method is executed and returns true. Therefore, the return value of `isOption` (line 4) does not change in comparison with the expected result. Since these methods have no other effects, the infection is not propagated.

```
1 public class DefaultParser {
2  private boolean isLongOption(String token) {...}
3
4  private boolean isOption(String token){
5   return isLongOption(token) || isShortOption(token); }
6
7  private boolean isArgument(final String token) {
8   return !isOption(token) || isNegativeNumber(token); }
```

```
9
10  private void handleToken(final String token) {
11  ...
12   else if (currentOption != null && currentOption.acceptsArg() && isArgument(token))
13   ...
14  }
15
16  public CommandLine parse(...){
17   ...
18   if (arguments != null) {
19     for (final String argument : arguments) {
20       handleToken(argument);
21     }
22   }
23   ...
24  }
25 }
```

Listing 4.5: An example of **no-propagation** in `commons-cli` where the execution masks the effects of the transformed method

A **no-propagation** occurrence might signal the need for code refactoring. In `commons-cli` we found that the body of `hasValueSeparator` declared in the `Option` class (line 1 in Listing 4.6) could be replaced by `return true` and, for the test cases in which the method should have returned false, the change in the program state does not reach the top level test code. It is interesting to notice that this is a public method, not directly assessed by any test case and it is used only by a private method in the same class. In the code of `processValue`, some specific actions are taken (lines 12 to 14) if the instance of `Option` has a value separator. If `hasValueSeparator` returns the wrong value, that is, the method returns true when it should have been false the effects of `processValue` are the same. The separator is not found in the given parameter (line 9), and the actions are never taken as the condition of the `while` loop in line 10 is false. So, in this example, `hasValueSeparator` has no effect whatsoever and the code could be refactored. Another way to solve the problem is to directly assess the method, as it is public.

```
1  public boolean hasValueSeparator() {
2    return valuesep > 0;
3  }
4
5  private void processValue(String value)
6  {
7   if (hasValueSeparator()) {
8     char sep = getValueSeparator();
9     int index = value.indexOf(sep);
10    while (index != -1) {
11     if (values.size() == numberOfArgs - 1) break;
12     add(value.substring(0, index));
13     value = value.substring(index + 1);
14     index = value.indexOf(sep);
15    }
16  }
17  add(value);
18 }
```

Listing 4.6: An example of a **no-propagation** symptom in `commons-cli` that might require refactoring

In general, a **no-propagation** symptom might be solved with the creation of new test cases which are closer, in the invocation chain, to the method in question. This increases the chances of immediate program infections to propagate to the top level code of the test case.

weak-oracle examples

When the infection is propagated to the test code we discover a **weak-oracle** symptom. The infection propagations are observed in the result value of the expressions in the code of the test

cases.

Listing 4.7 exposes an example of a **weak-oracle** symptom in `jcodemodel`.

The `annotate` methods of the class `JPackage`, declared in lines 5 and 15 have both a such a symptom. The body of these two methods can be replaced by `return null` and the test suite executes without noticing the change. These methods are executed by the `testPackageAnnotation` test case shown in line 27. This test seems to be designed to actually verify the package annotation functionality. Both transformations do infect the program state, as they make the method return null, which is not the expected value. The infection reaches the test code in line 29, where the method is invoked and its result could be directly asserted. The transformation to the method in line 15 can be also observed in line 30. This method has a side effect on the `JCodeModel` instance by adding a reference to the annotation class. In the `JCodeModel` class, these references are stored in the `m_aRefClasses` field (line 22). Our process notices that this field does not have the expected size when the transformation is executed, as it misses a reference to `Inherited`. The oracle in the same line 30 only checks if the code model is syntactically correct. It does not check if the package has been annotated.

```java
public class JPackage ... {
  private final JCodeModel m_aOwner;
  private List<JAnnotationUse> m_aAnnotations;

   public JAnnotationUse annotate (AbstractJClass aClazz) {
      JCValueEnforcer.isFalse(isUnnamed (), "...");
      if (m_aAnnotations == null) {
        m_aAnnotations = new ArrayList<>();
      }
      JAnnotationUse a = new JAnnotationUse(aClazz);
      m_aAnnotations.add(a);
      return a;
   }

  public JAnnotationUse annotate (Class aClazz) {
    return annotate(m_aOwner.ref(aClazz));
  }
  ...
}

public class JCodeModel {
  private Map<Class<?>, JReferencedClass> m_aRefClasses = new HashMap<> ();
   ...
}

@Test
public void testPackageAnnotation () {
  JCodeModel cm = new JCodeModel();
  cm._package("foo").annotate(Inherited.class);
  CodeModelTestsHelper.parseCodeModel(cm);
}
```

Listing 4.7: Example of **weak-oracle** symptoms in public methods in `jcodemodel`

4.3.5 The role of testability

Testability issues require modifying or refactoring the application code to be solved. We inspected the 58 transformations of the first nine projects in Table 4.1. We manually wrote a test case to detect each transformation. We found only two cases where the application code must be modified to address the symptom, one of them is exposed in Listing 4.8 and the other in Listing 4.9. We now discuss both findings.

Listing 4.8 shows the example of a private void method named `getFromCache` in `gson-fire`. It is used by a public method (lines 16 and 21) to retrieve a cached result. Under extreme transformations the instructions of this method are replaced by `return null`. The immediate program state is infected for some invocations of the method where the result value should be non-null.

When the cached result is null, the actual value is computed once again, therefore the returned value of the public method is the same. The effects are propagated and observed in the `cache` field (line 24). Under the transformation no value is ever cached and the collection remains empty when it should contain some elements. However, there is no way to verify the effects of this method from the test code.

Reneri is able to get the size of `cache` via reflection, but the field is not accessible. In order to catch the extreme transformation the test code needs to check the content of `cache`. The only way to achieve this is to modify the code of the class by adding a new accessor or changing the visibility of `cache`.

```
1  public abstract class AnnotationInspector {
2    private ConcurrentMap cache = new ConcurrentHashMap();
3
4    private Collection getFromCache(Class clazz, Class annotation) {
5      Map annotationMap = cache.get(clazz);
6      if(annotationMap != null){
7        Collection methods = annotationMap.get(annotation);
8         if(methods != null)
9           return methods;
10     }
11     return null;
12  }
13
14   public Collection getAnnotatedMembers(Class clazz, Class annotation){
15     if(clazz != null) {
16       Collection members = getFromCache(clazz, annotation);
17       if (members != null) {
18         return members;
19       }
20       //Cache miss
21       members = getFromCache(clazz, annotation);
22       if (members == null) {
23         ...
24         ConcurrentMap storedAnnotationMap = cache.putIfAbsent(clazz, newAnnotationMap);
25         ...
26       }
27    }
28    return Collections.emptyList();
29   }
30 }
```

Listing 4.8: Example of a testability issue preventing the solution of a **weak-oracle** symptom in `gson-fire`

Listing 4.9 shows `isParseable`, a private static boolean method from `yahoofinance-api`. This method is used by nine other public methods that follow the same pattern as the one shown in line 7. If the input value can not be parsed as per the result of `isParseable`, `getBigDecimal` returns a default value. The default value is also returned if the actual parsing throws an exception (line 15).

Some test cases in the existing test suite use `"N/A"` as input for `getBigDecimal`. In the execution of these test cases `isParseable` returns false. When the body of `isParseable` is changed to `return true` no test case notices the change. What happens is that, when the method is supposed to return false, as in the case of `"N/A"`, the actual parsing (line 14) throws an exception, which is captured in the body of `getBigDecimal` (line 15) and then the method returns the default value. So `getBigDecimal` ends up returning the same value it is expected to return. The effects of the extreme transformation are able to infect the program state at the end of the invocation of `isParseable`, as it returns true instead of false. However the effects are not propagated to `getBigDecimal` that returns the expected value.

So, no matter the outcome of `isParseable`, the result of `getBigDecimal` is the same. Since `isParseable` is private it can not be tested directly. Even when the program state is infected when the transformation is executed by the test suite, the effects do not propagate any further. As there is no other effect than the result value, an error introduced inside `isParseable` can not be observed.

```java
public class Utils {
  private static boolean isParseable(String data) {
    return !(data == null || data.equals("N/A") || data.equals("-")
           || data.equals("") || data.equals("nan"));
  }

  public static BigDecimal getBigDecimal(String data) {
    BigDecimal result = null;
    if (!Utils.isParseable(data)) {
      return result;
    }
    try {
      ...
      result = new BigDecimal(data).multiply(multiplier);
    } catch (NumberFormatException e) {
      log.debug("Failed to parse: " + data, e);
    }
    return result;
  }
}
```

Listing 4.9: Example of a non-testable method in `yahoofinance`

We observed only two testability issues among 58 undetected extreme transformations. This may indicate that the lack of testability does not appear to be a major influence for extreme transformations to pass unnoticed by the test cases.

Answer to RQ2

42% of the transformations infecting the program state, propagate to an observable point. In general, methods with **weak-oracle** symptoms are closer to the test cases than **no-propagation** in the invocation sequence. Transformations that do not propagate may require new test cases closer to the method. Testability issues may prevent **weak-oracle** symptoms to be solved without refactoring.

4.3.6 RQ3: *Are the suggestions synthesized by Reneri valuable for the developers?*

With **RQ3** we perform a qualitative evaluation of the suggestions synthesized by Reneri. We want to know if developers find these suggestions relevant and helpful to fix undetected transformations. To answer this question we target four open-source projects for which we are able to directly consult developers who have a strong knowledge about the code.

For each project we select up to six undetected extreme transformations for which Reneri generated a suggestion. We manually select these transformations, so that they represent interesting testing cases for the developers. Table Table 4.3 shows the projects and the symptoms of the selected issues.

For each issue we create a form containing (i) a link to the automatically generated report (similar to those shown in Figures 4.1, 4.2 and 4.3); and (ii) a set of questions.

The questionnaire can be consulted in Appendix C.

Qualitative feedback from the developers

Table 4.4 summarizes the developers' feedback. We divide the answers from the developers according to the three types of symptom. Of the 23 issues that were discussed, 18 were considered as relevant or of medium relevance. The developers considered the report to contain helpful information in 18 cases, and even thought that the exact testing solution was given in 4 cases. In all cases, the developers, could emit a verdict about all the testing issues in around 30 minutes.

Table 4.3: Projects involved in the empirical validation with developers

Project	Issues	**ni**	**np**	**wo**
`Funcon4J`	5	2	3	0
`greycat`	6	2	2	2
`sat4j-core`	6	2	2	2
`xwiki-commons-job`	6	2	3	1
Total	23	8	10	5

Table 4.4: Summary of the feedback given by developers

	ni	**np**	**wo**	Total
The issue in the description is:				
relevant	4	5	2	11
of medium relevance	2	4	1	7
not important	1	1	1	3
not really a testing issue		1	1	2
The suggestion provided:				
points to the exact solution	3		1	4
provides helpful information	4	7	3	14
is not really helpful	1	2		3
is misleading		1	1	2
Developers solve the issue by:				
adding a new assertion	1	4	1	6
slightly modifying a test case	1			1
creating a new test case	3	1	1	5
performing other actions	3	5	3	11

Developers considered that six transformations can be solved with the addition of an assertion, only one by slightly modifying an existing test case and five with the creation of a new test case. In 11 cases developers considered that the transformations should be solved by other actions. These actions included, for example, a complete replacement of the assertions in a test case, or a combination of new input and new assertions, or, instead of adding a new verification to an existing test case, developers would prefer the create a new test case by repeating the same code with the new assertion. The solutions are in fact influenced by the testing practices of each developer.

Even when developers would not modify the test code as suggested, the report provides information that is generally considered as helpful and the proposed solution as a valid starting point to ease the understanding of the testing issue.

The developers found that the synthesized suggestion was not helpful in three cases, two **no-propagation** and one **no-infection**. In two of those cases, the developers argue that the suggestion should contain more information about the state differences between the original and the transformed method. For example, developers would like the tool to identify the instruction in the test code that triggered the method invocation for which the state difference was observed. The third case was a method related to the performance of the code. The developer argued that a test case for this method would be "artificial".

Two suggestions were considered as misleading. In one case, the issue could not be reproduced by the developer. In the other, the developer could not make the connection between the program state difference and the test cases executing the method. So, developers find the suggestions not helpful or misleading when they fail to understand what caused the program state difference between the executions of the original and the transformed method. Suggestions could be further improved with additional information, for example, the stack trace containing the invocation sequence from the test code to method in **no-propagation** symptoms.

Actual solutions given by the developers

Developers actually solved 13 of the issues with 11 commits. The details of the commits are available online [7]. One issue in `funcon4j` was not solved as it was related to a testability problem. No issue was fixed for `greycat` due to non-technical and unrelated reasons. In this section we provide two examples of how developers solved the extreme transformations and how the solutions relate to the synthesized suggestion.

In project `funcon4j` the developer was presented with a **no-infection** symptom. The suggestion, was to create a variant of the existing test case to make the method produce a different value. In the questionnaire, the developer answered that the issue can be solved with the addition of a new assertion. Listing 4.10 shows the actual test modification: the addition of line 2. The commit can be seen at https://github.com/manuelleduc/Funcon4J/commit/63722262313fb2dac5b516bbae5f04e0502e7f26.

The developer added a new assertion but she also included a new input derived from the existing code, using a small modification. The actual test improvement corresponds to the suggestion generated by Reneri.

```
1 test("{a = 1, b = 2} > {b = 1, a = 1};;","true");
2 test("{a = 1, b = 1} > {b = 1, a = 1};;","false"); // Fix
3 test("\"abc\" > \"abd\";;","false");
```

Listing 4.10: Example of a test fix created by a developer based on the information contained in the Reneri report

In Listing 4.11 we show an example of a test code fix created by a developer of `sat4j-core`, with the help of Reneri. The original test code is in lines 2 and 8. The report, available at https://github.com/STAMP-project/descartes-amplification-experiments/blob/master/validation/sat4j-core/selected-issues/6.md pointed at line 6. The extreme transformation made the method invocation produce a null value, while the original code should produce a non-null array. The test fix can be seen between lines 11 and 21. The developer confirmed that Reneri provided the exact solution and added the proposed assertion (cf. 18). An extra assertion was also added in the following line.

[7] https://github.com/STAMP-project/descartes-amplification-experiments/blob/master/validation/commits.md

```java
// Original code
clause.push(-3);
solver.addClause(clause);
int counter = 0;
while (solver.isSatisfiable() && counter < 10) {
  solver.model();
  counter++;
}

// Fix created by the developer
clause.push(-3);
solver.addClause(clause);
int counter = 0;
int[] model;
while (solver.isSatisfiable() && counter < 10) {
  solver.model();
  model = solver.model();
  assertNotNull(model); //Fix
  assertEquals(3, model.length); //Fix
  counter++;
}
```

Listing 4.11: Example of the addition of an assertion guided by the generated report

Answer to RQ3

Our empirical evaluation shows that the synthesized suggestions are helpful for developers. In the best case, Reneri even points to the exact solution. The real test fixes created by the developers to solve the testing problems are in line with the suggestions synthesized by Reneri.

4.3.7 RQ4: *Can developers leverage test improvement tools to deal with undetected extreme transformations?*

With this question we explore if state-of-the-art test generation tools can help developers to deal with undetected extreme transformations, *i.e.* they can generate tests that detect the extreme transformations missed by the original test suite.

With current tools, we have two possibilities: generate test cases from scratch targeting the methods in question; or, improve existing test cases that are known to reach the method in the extreme transformation. The expected benefit of the first alternative is that test cases are as close as possible to the method. This increases the chances to observe the state differences. Also, some developers prefer to preserve existing test cases (cf.4.3.6). The second option, uses existing test cases as seeds for the generation process. These tests already have inputs able to execute the method in the transformation. These test cases may just need small code adjustments to make the method return a different value or to actually verify the effects of the transformation.

Based on the aforementioned options, we implemented two improvement strategies. One strategy is based on EvoSuite [27], a state-of-the-art tool for automatic test generation in Java. It generates test cases from scratch and uses coverage and weak mutation to assess the generated test suites. The other strategy is based on DSpot [14], is a test improvement tool that can use the ratio of undetected extreme transformation as the objective function.

We applied both strategies to the projects listed in Table 4.1. Since both tools produce non-deterministic results, we attempted the improvement process for each project and strategy ten times.

We consider an undetected extreme transformation to be *solved* by one strategy if it is detected by the test cases generated in any of the improvement attempts. In our experiments we evaluate both strategies in terms of the stability of the results, that is, if they obtain similar results in all attempts. We also analyze the absolute number of transformations solved by each strategy in total and considering the type of symptom.

In this section, we briefly summarize how each strategy selects the input and configures the tools. Then we discuss the results we obtained and their implications.

Detecting extreme transformations with EVOSUITE

EVOSUITE is a test generation tool for Java classes. It implements a search-based approach to produce test suites from scratch. As a fitness function, EVOSUITE maximizes a combination of coverage criteria and weak mutation score. EVOSUITE also minimizes the generated test suite to keep only test cases that are valuable according to the fitness function. The tool adds assertions to the test cases based on the observation of the test executions.

In our experiment we identified a set of methods for each project as targets for EVOSUITE. The set was computed as follows: (i) We include all methods with at least one undetected extreme transformation (ii) If there is a private method in the set, we remove it and add all the accessible methods in the project that could be used instead. This is done in the same way as in stages 2 and 3 of Reneri. (iii) If there is a method declared in an abstract class or an interface (default interface methods in Java 8+), then we remove the method from the targets and add all its implementations instead. (iv) The two previous steps are repeated until no further change can be done to the set.

The second column in Table 4.5 shows the number of target methods identified for each study subject. Recall that the goal is to create test cases specifying the methods with undetected extreme transformations using these identified target methods as entry points.

Next, we provide the target methods as inputs for EVOSUITE. We use the default parameter configuration for the tool. The outcome, for a specific project, is a set of test cases generated for all the target methods in that project. As final step, we run the generated test cases to determine if the EVOSUITE tests can detect the extreme transformations that were previously undetected.

Detecting extreme transformations with DSpot

DSpot is a tool designed to improve test cases written by developers. It takes as input existing test cases and gradually improves their fault detection capabilities. DSpot transforms the input of the test cases by altering literals or adding and removing method calls. Then, new regression assertions are inserted to the modified test cases. The assertions are built using the values observed through existing getter methods. DSpot only keeps the generated test cases that increment the mutation score of the existing test suite. As a proxy for the mutation score, we used the ratio of detected extreme transformations in the code covered by the test to improve.

It is prohibitively expensive to use DSpot and improve all test cases executing one method and do the same for every method with undetected extreme transformations. Recall that some of these methods are executed by more than 100 test cases (cf. Table 4.1). For this reason, we instruct DSpot to improve the test case that is the closest, in a sequence call, to a method where there is an undetected extreme transformation.

We detect the closest test case by executing the test suite and obtaining the stack trace from any method in a test class to the application methods with undetected extreme transformations. For each application method, the closest test case is the test method that produces the shortest stack trace. Different methods may be executed by the same test cases. In some situations, the closest test case to more than one method is the same. In such a situation we keep the same test case for these methods.

The selection results in a set of test cases for each project. The third column in Table 4.5 shows the number of test cases identified for each study subject.

An attempt to solve the undetected extreme transformations for a given project, consists in executing DSpot for each test case in the identified set. We used all DSpot search operators (amplifiers) available (by default DSpot only adds regression assertions to existing test cases) and performed three amplification iterations for each target. The output of one attempt is the combination of all the improved test cases.

Experimental Results

Figure 4.6 shows the distribution of the relative improvement obtained for each project. The y-axis is the percentage of undetected extreme transformations that the new test cases can detect.

Table 4.5: Targets identified for automatic test generation and test improvement. The *Target methods* column shows the methods identified as targets to be used with EVOSUITE. The *Target test cases* are the test cases selected for improvement with DSpot

Project	Target methods	Target test cases
jpush-api-java-client	2	2
commons-cli	6	3
jopt-simple	2	3
yahoofinance-api	11	1
gson-fire	7	3
j2html	3	2
spring-petclinic	6	5
javapoet	10	10
eaxy	21	8
java-html-sanitizer	13	11
cron-utils	28	10
commons-codec	30	18
jsoup	60	26
TridentSDK	49	13
jcodemodel	147	25
Total	395	140

The figure also shows the number of extreme transformations detected for the worst and best attempts.

Both strategies produce test improvements for 11/15 projects. Eight projects are improved by both strategies. No strategy can generate tests that fix undetected extreme transformations in `jpush-api-java-client`. DSpot produces the best improvement in one attempt for seven projects while EVOSUITE achieves the best improvement for six projects. The same best improvement is obtained for `java-html-sanitizer` and `jpush-api-java-client`.

No strategy reaches an improvement beyond 85% in one attempt. DSpot reaches this percentage by handling 6/7 extreme transformations in `jopt-simple`. The EVOSUITE strategy achieves an 83% improvement handling 5/6 extreme transformations in `spring-petclinic`. In the largest project, `jcodemodel`, the EVOSUITE strategy handles 64/118 transformations while DSpot handles 89/118. The improvement is below 75% for both cases.

This means that no strategy was considerably better than the other in terms of handling all projects and even all the transformations on each project.

Inspecting the improvement distributions we notice that DSpot has better convergence in all projects but `jcodemodel` and `jsoup`. This is a consequence of using the existing test cases as seeds for the generation process. Those test cases have been created by developers, therefore, they are close to a local optimum and DSpot, as it implements a local search approach, is able to converge more frequently to this solution.

We computed the set of solved extreme transformations for each strategy. That is, the extreme transformations detected in at least one attempt in one project by each strategy. EVOSUITE solves 146 (47%) of all 312 transformations. On its side, the DSpot strategy solved 180 (58%). Only 87 (28%) transformations are solved by both strategies. A total of 239 (77%) are solved by any of the strategies, which leaves 73 (23%) transformations unsolved.

Figure 4.7 shows a Venn diagram displaying the relationship between the sets of extreme transformations solved by both tools. The set of unsolved transformation is also provided as a reference.

DSpot solves more extreme transformations, but no strategy solves more than 60% of all transformations. The intersection between the transformations solved by both strategies is low, below 30%. Only when both results are combined, we are able to solve 77% of all transformations but still remains below 80%. This is a direct result of the very diverse nature of undetected extreme transformations. A subset of the transformations is more challenging for one of the strategies than the other.

Table 4.6 shows the set of solved extreme transformations by symptom (It is only a coincidence that the intersection for all symptoms is 29). DSpot performs better for all symptoms but more

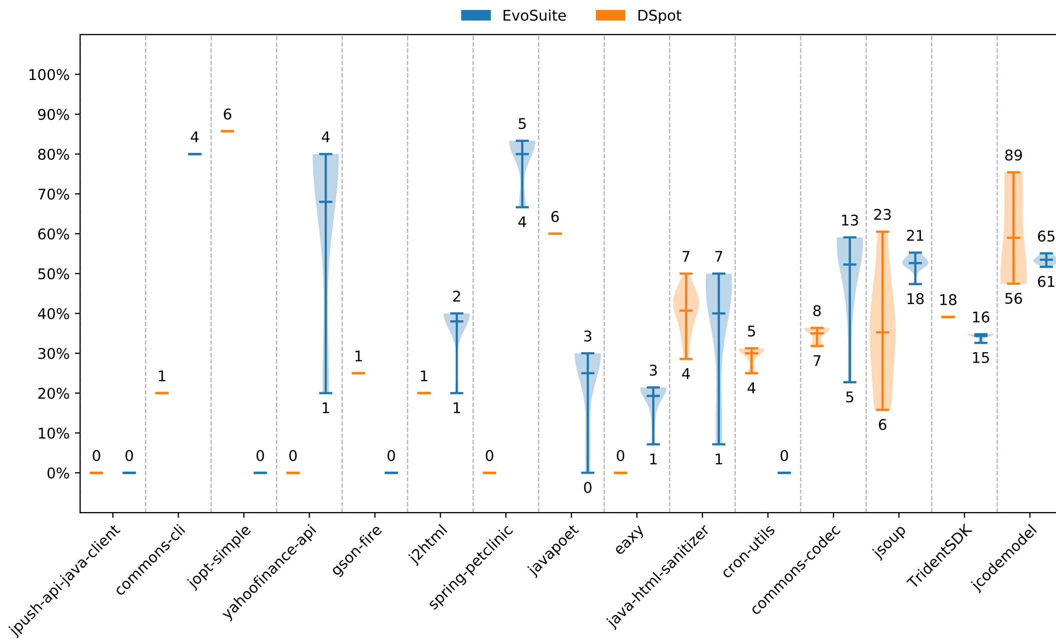


Figure 4.6: Automated test improvement for 15 projects

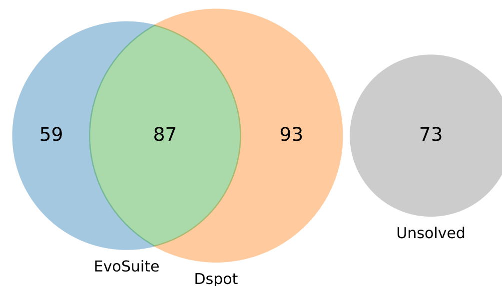


Figure 4.7: Relationship between the undetected extreme transformations solved by the strategies based on both tools and the unsolved transformations

Table 4.6: Solved transformations according to their symptom. Second and third columns show the transformations solved by EVOSUITE and DSpot respectively. *Intersection* shows the number of transformations solved by both tools and *Union* shows those transformations solved by at least one strategy.

Symptom	EVOSUITE	DSpot	Intersection	Union
no-infection	53	57	29	81
no-propagation	55	66	29	92
weak-oracle	38	57	29	66
Total	146	180	87	239

notable for **weak-oracle** symptoms. These are precisely the symptoms that Reneri identifies that benefit the most from modifying existing test cases, in particular, the addition of new assertions, which is one of the main features of DSpot.

DSpot produces better results because it uses the test cases written by developers as seed and its fitness function considers extreme tranformations. We believe that this strategy is closer to what a developer would do. The developer actions exemplified in 4.3.6 provide supporting evidence in this sense. However, the strategy based on EVOSUITE does not fall too much behind. A better fine-tuning of the tool and the incorporation of a notion of extreme transformation to the fitness function may help the strategy achieve a much better result.

This opens the opportunity for a more targeted solution that fully exploits the results of Reneri. Nevertheless, we observe in 4.3.6 that developers have strong and diverse opinions when it comes to refactor the testing code to deal with extreme transformations. Providing them with helpful and well localized information extracted from dynamic and static analysis and perhaps small code changes might be the adequate solution instead of a fully automated approach.

Examples

In this section we illustrate, with two examples, scenarios in which only one of the strategies was successful.

Listing 4.12 shows the `conditionC0` method in line 3. This is a private method of a class in `commons-codec`. The body of the method can be replaced by `return false` and no test case fails. Notice that this method is reachable through a sequence of at least three invocations comprising non-trivial conditions. So, generating an input that even makes the execution reach the method is already difficult.

```
1  public class DoubleMetaphone ... {
2
3    private boolean conditionC0(final String value, final int index) {
4      if (contains(value, index, 4, "CHIA")) {
5        return true;
6      } else if (index <= 1) {
7        return false;
8      } else if (isVowel(charAt(value, index - 2))) {
9        return false;
10     } else if (!contains(value, index - 1, 3, "ACH")) {
11       return false;
12     } else {
13       final char c = charAt(value, index + 2);
14       return (c != 'I' && c != 'E') ||
15       contains(value, index - 2, 6, "BACHER", "MACHER");
16     }
17   }
18
19   private int handleC(final String value, final DoubleMetaphoneResult result, int index) {
20     if (conditionC0(value, index)) { // very confusing, moved out
21       result.append('K');
22       index += 2;
23     }
```

```
24     ...
25   }
26
27   public String doubleMetaphone(String value, final boolean alternate) {
28     ...
29     while (!result.isComplete() && index <= value.length() - 1) {
30       switch (value.charAt(index)) {
31         ...
32         case 'C':
33           index = handleC(value, result, index);
34           break;
35         ...
36       }
37       ...
38     }
39     ...
40   }
41
42   public String doubleMetaphone(final String value) {
43     return doubleMetaphone(value, false);
44   }
45 }
```

Listing 4.12: An extreme transformation in `conditionC0` was solved by EvoSuite

To solve this transformation, DSpot is given as input a test case whose code is composed of statements like the ones shown in Listing 4.13. These instructions correspond to custom assertions verifying the output for an specified input. In front of such a test case, DSpot starts to randomly alter the literal values in the test code. This random process did not produce anything of value to solve the transformation.

```
1 ...
2 assertDoubleMetaphoneAlt("MKFR", "MacCafferey");
3 assertDoubleMetaphoneAlt("STFN", "Stephan");
4 ...
```

Listing 4.13: Extract of the test case given as input to DSpot

On its side EvoSuite generated the test case shown in Listing 4.14. It creates the required instance and triggers the required behaviors. An interesting fact to notice here is that the generated test case is effective only because the tool reused as input a string (line 15) that appeared in the code of the `conditionC0` method (Listing 4.12 line 2).

```
1 DoubleMetaphone doubleMetaphone0 = new DoubleMetaphone();
2 String string0 = doubleMetaphone0.doubleMetaphone("MACHER");
3 assertNotNull(string0);
4 assertEquals("MKR", string0);
5 assertEquals(4, doubleMetaphone0.getMaxCodeLen());
```

Listing 4.14: Solution given by EvoSuite

Listing 4.15 shows the `hasHeaderComment` method from `jcodemodel`. The body of this method can be replaced by `return false` and no test case would notice the change. The method itself is very simple, but its result depends on the state of the receiver instance which can be only modified by invoking `headerComment` (line 9). So, to being able to solve the transformation, the test case must invoke `headerComment` first and then `hasHeaderComment` to make the latter method return true.

```
1 public class JDefinedClass ... {
2   private JDocComment m_aHeaderComment;
3
4   public boolean hasHeaderComment ()
5   {
6     return m_aHeaderComment != null;
7   }
8
```

```
9    public JDocComment headerComment ()
10   {
11     if (m_aHeaderComment == null) {
12       m_aHeaderComment = new JDocComment (owner ());
13     }
14     return m_aHeaderComment;
15   }
16   ...
17 }
```

Listing 4.15: An extreme transformation in `hasHeaderComment` was solved by DSpot

In all attempts EVOSUITE failed to produce a test case with the right method sequence call. On its side, DSpot was given as input the test case shown in Listing 4.16 and produced the test case shown in Listing 4.17.

```
1 final JCodeModel cm = new JCodeModel ();
2 final JDefinedClass jClass = cm._class ("EmptyNarrowed", EClassType.INTERFACE);
3 final AbstractJClass hashMap = cm.ref (java.util.HashMap.class).narrowEmpty ();
4 jClass.field (JMod.PRIVATE, cm.ref (Map.class).narrow (String.class), "strMap",
      JExpr._new (hashMap));
5 CodeModelTestsHelper.parseCodeModel (cm);
```

Listing 4.16: Input given to DSpot

Notice how, in the resulting test case, an invocation to `headerComment` is inserted in line 5. Later in the code `hasHeaderComment` is used by DSpot to observe the object and thus capturing the transformation.

```
1 final JCodeModel cm = new JCodeModel();
2 final JDefinedClass jClass = cm._class("EmptyNarrowed", EClassType.INTERFACE);
3 final AbstractJClass hashMap = cm.ref(HashMap.class).narrowEmpty();
4 JFieldVar o58_8 = jClass.field(JMod.PRIVATE, cm.ref(Map.class).narrow(String.class),
      "strMap", JExpr._new(hashMap));
5 JDocComment o58_13 = jClass.headerComment();
6 ...
7 assertTrue(jClass.hasHeaderComment());
8 ...
```

Listing 4.17: Solution obtained with DSpot

As seen in the previous examples, the effectiveness of each tool is influenced by both, the application code and the test code.

Answer to RQ4

DSpot solves 58% of the undetected transformations, while the EVOSUITE solves 47% . None of state-of-the-art test generation tools can fix all testing issues revealed by extreme transformations.

4.4 Threats to validity

As said before, we believe that the definitions included in Section 4.2.1 and the entire process implemented in Reneri can be extrapolated to other programming languages and runtimes. However we can not assure that the results we have obtained for **RQ1** and **RQ2** may generalize to other environments and even a different set of Java projects. The set of study subjects used to answer these two questions include only 15 projects, which is not large enough to derive statistically robust results. This selection is also restricted to small to medium sized Maven projects with a single module. In large and multimodule projects the situation may be different. Nonetheless, our set of subjects is diverse with regards to the obtained results.

The tooling we have developed and also the external tools we have used are not exempt from the presence of bugs. Reneri has been conceived as an open-source project available in Github.

We have used PITest 1.4.7, and Descartes 1.2.5 to answer all questions. We used DSpot 2.1.0 and built EVOSUITE from revision 1895b6d to answer **RQ4**. Using a different version of these tools may produce a different result given their natural evolution.

The validation to answer **RQ3** was conducted over a very small set of projects and issues. Also, the selection included projects whose developers we know and can reach. In order to be able to generalize these results a more impartial and larger evaluation is required. However, we selected projects with different application domains and developers with different backgrounds and experience.

4.5 Conclusion

In this chapter we describe an infection-propagation analysis to generate actionable suggestions that can help developers deal with undetected extreme transformations. The process is implemented in an open-source tool, Reneri, that can target Java projects built with Maven. With the help of Reneri we study the undetected extreme transformations of 15 Java open-source projects. In 312 transformations:

- 37% do not infect the immediate program state
- 37% infect the immediate program state but the infection do not propagate to an observable point
- the rest propagate the infection to an observable point, yet the existing test cases do not assess the program state items that are affected.

With Reneri we addressed the fourth and final objective of this thesis. We validated the suggestions generated by Reneri with the help of the developers of three open-source projects. Most suggestions provided helpful information or the exact solution to detect the extreme transformations.

Part of our results supports the observations made by Niedermayr and Wagner [52]. These authors showed a correlation between the stack distance from the test code to the pseudo-tested method and the test effectiveness. In this chapter we provide evidence that the effects of methods with higher stack distance are more likely to be masked in the invocation sequence. The core of our contribution is completely novel: automatically analyze undetected extreme transformations to generate suggestions to be used in improving the test suite.

In this chapter we also explore two automatic strategies to solve undetected extreme transformations. These strategies were based in state-of-the-art test generation and test improvement tools. The DSpot based strategy produces better results as it uses the ratio of undetected extreme transformations as fitness function and a seed that already executes the method involved in the transformation.

CONCLUSION

The main objective of this thesis is to help developers in the assessment of the quality of their test suites and automatically generate concrete test improvement suggestions. We have explored extreme transformations as the vehicle to detect testing issues.

We have built two main tools: Descartes, a PITest extension to challenge test suites against extreme transformations and discover pseudo-tested methods; and Reneri, which implements an infection-propagation analysis to generate suggestions for test improvements.

In Chapter 2 we presented Descartes as a robust implementation for extreme transformations and reported its industrial adoption. We also compared the use of extreme transformations to traditional mutation testing in terms of execution time and number of mutants in 21 open-source projects. Extreme transformations create less than 30% of the program variants than the traditional approach. The gain in execution time is also notable. We saw that the detection pseudo-tested methods is able to spot relevant testing issues. However, extreme transformations produce coarse-grained results.

As per the results of the same Chapter 2, extreme transformations are not a substitute of traditional mutation testing. They are more a complement. Projects in their earlier stages or with low-quality test suites may benefit the most from the outcome of a tool like Descartes. These projects could first work on the issues that can be discovered with extreme transformations before using mutation testing with more fine-grained operators. Although, we have shown that a mature project with strict testing practices can also improve their existing test cases with the help of Descartes.

In Chapter 3 we performed an in-depth analysis of pseudo-tested methods to check whether they can inform developers about the quality of their test cases. In this analysis we mitigate the threats to the validity of the results of Niedermayr and colleagues with a different set of projects and different tooling. We found pseudo-tested methods in all the 21 projects we studied on ratios from 1% to 46% of all inspected methods. We confirmed, using the traditional mutation score, the fact that pseudo-tested methods are the weakest point in the project in terms of fault detection capabilities.

We provided evidence that pseudo-tested methods hint the presence of weak test oracles, which was confirmed by developers through the acceptance of pull requests made by us. Also, through the interaction with developers we were able to understand the characteristic of pseudo-tested methods they consider more relevant and worth further testing effort. In general, methods involved in core functionalities, widely used in the code, methods used by external clients and those verifying preconditions are among the ones developers give more importance.

Even when the effects of extreme transformations are easier to understand when compared to a traditional mutant, they require a deep insight on how the application code and the test suite interact. Only through this insight developers are able to wrasp their importance and properly solve the related testing issues.

In Chapter 4 we presented an infection-propagation analysis to produce test improvement suggestions for developers. This analysis investigates undetected extreme transformations to discover the parts of the program state in which they can be noticed during test executions. We applied this analysis to 15 open-source projects. We observed that 37% extreme transformations do not infect the immediate program state, 37% infect the program state but the infection do not propagate to an observable point, the rest propagate the infection to an observable point, yet the existing test cases do not assess the program state items that are affected. We evaluated the suggestions Reneri generates by consulting developers from three open-source projects. Most suggestions provided helpful information or the exact solution to detect the extreme transformations.

In the same Chapter 4 we showed evidence that the effects of methods with higher stack distance are more likely to be masked in the invocation sequence and therefore not propagated to the test code. This corroborates the results of Niedermayr and Wagner who showed a correlation between the stack distance and test effectiveness.

In general we can conclude that extreme transformations can be leveraged to improve a test suite. They can point to important testing issues and, by performing dynamic analyses, it is pos-

sible to find concrete actions to improve the test suite. However, this line of work is far from being finished. In the next section we discuss potential future research directions, their perspectives and how the tools developed in this work can be improved.

Future work and perspectives

The results we have presented in this thesis open the perspectives to continue studying extreme transformations and to find newer ways to support test improvements. The two main tools we have built can be evolved with new functionalities and better analyses. Here we discuss the lines of work that can be followed in the future for the assessment of test cases and the generation of improvement suggestions.

Detection of test weakness

The category of pseudo-tested methods, as currently defined, can be quite restrictive. All undetected extreme transformations may lead to a potential testing issue. Methods with mixed results, that is, with detected and undetected extreme transformations at the same time, are not considered as pseudo-tested. Yet we have observed, and provided examples, that these methods can also point to testing issues. For example, boolean methods returning only true in the execution of the test suite will exhibit this behavior. So, these methods need to be separated from those where all transformations are detected. Diagnosing the testing issue in those methods should consider undetected and also detected extreme transformations. At the moment, Descartes simply informs developers about these methods.

Petrovic *et al* [60], observed that there are notable differences among different programming languages when applying mutation testing. This may be also the case for extreme transformations. A future line of research could expand the study of extreme transformations and pseudo-tested methods to other languages. Other authors [74] have taken some steps in this direction by exploring these concepts in Python.

It is imperative for developers to know which are the findings that deserve their immediate attention. At the moment, we give the same importance to all pseudo-tested methods and extreme transformations. A ranking of these methods would be useful for practitioners. An initial sorting criterion that could be explored is to consider the static call graph of the project. Methods which are used the most should have more relevance. Also, the more a method depends on others, the lower its relevance should be. The extreme case are delegation patterns, which are automatically skipped by Descartes. Methods adding little code on top of invocations to other methods should have a lower rank than those they invoke. This is supported by the feedback we obtained from developers in Chapter 3.

Kurtz *et al* [43] explore the role of redundant mutants in the assessment of the quality of the test suite. They compute a subsumption graph between traditional mutants with the help of the entire killing matrix, that is, challenging each mutant against each test regardless of whether they are detected by an earlier test case. Each mutant is characterized by the set of test cases able to detect it. A mutant is said to subsume the other if all test cases detecting the former mutant also detect the latter. A better mutation score results from only considering non-subsumed mutants. We explored the obtention of a full killing matrix for extreme transformations in 13 open-source projects from the ones studied in Chapter 3. We observed, though not quantified, that there is redundancy among extreme transformations. This redundancy could be studied to assess its practical consequences. A redundancy analysis can help, for example, to determine the most relevant undetected extreme transformations.

Improving Descartes

Descartes, as PITest, transforms the bytecode of compiled Java classes. This opens the door for the application of Descartes to projects developed in other JVM languages like Kotlin or Scala. The tool should avoid applying unproductive transformations in order to make the results accurate for the semantics of those languages. Some preliminary steps have been taken towards this goal. For example Descartes does not force a `return null` transformation in methods annotated by the

Kotlin compiler as `@NotNull`. However, more insight about the languages, their recommended practices and compilers is required to provide a comprehensive solution.

The extensible architecture of PITest makes possible the development of Descartes. Our extension brings a new set of mutation operators. Building a new mutation engine facilitated the development of Descartes as a separate product from PITest. However, this prevents the combined use of Descartes and Gregor, the default mutation engine. The operators from both engines can not be used at the same time. Also neither Descartes nor Gregor are extensible on their own. A way to overcome this limitation is to build a new mutation engine that can be dynamically extended with new mutation operators. There are two non-exclusive options to achieve this. One approach could be to configure the mutation engine using a Domain Specific Language to specify mutations, similar to what Major does [38]; the other is to use a plugin architecture based on reflection or a similar mechanism. This mutation engine shall contain by default all operator form Gregor and Descartes.

Suggestions for test improvement

Extreme transformations eliminate all the effects of the methods. This has advantages but, as we saw in Chapter 2, makes the analysis coarse-grained and limited. Deletion mutation operators [75] seem to provide a trade-off between traditional operators and extreme transformations. A future line of work could replicate the analysis performed by Reneri with deletion operators and validate the suggestions that can be obtained from them. This can also help to evaluate the testability of the code, as defined by Voas [81].

The suggestion generation process described in Chapter 4 and implemented by Reneri use extreme transformations. However, the process is independent from these transformations and could be used directly to analyze traditional live mutants. A future study can explore whether this analysis can produce meaningful suggestions for developers and if it can help to ease the understanding of live mutants.

Developers do not tend to improve the quality of the tests verifying pseudo-tested methods. Either they are not aware of the testing issue or they do not consider these methods important enough for the effort. We explored eight projects from the ones included in Chapter 3. In these projects we studied the last 100 commits that edited code files and not only data or configuration files. We selected all methods categorized as pseudo-tested in at least one of the commits. This resulted in a set of 1189 methods. Only 20 of these methods improved, that is, they went from being pseudo-tested to be required. Three methods had erratic category changes, that is, they changed several times from pseudo-tested to required and back. Further studies are required to generalize these partial results. However, this situation opens the opportunity to automated solutions. It could be interesting to validate with developers the inclusion of the test generated with DSpot and EVOSUITE, as done in Chapter 4, as a complement to the suggestions provided by Reneri.

Improving Reneri

Reneri generates suggestions from extreme transformations that consist in very well localized issues. In the case of weak-oracles we can identify the exact location where the test code may be modified to catch the extreme transformation. A next step is to include Reneri in an Integrated Development Environment as it was done for ReaAssert [16]. The suggestions generated by Reneri could be displayed as warnings in the code.

At the moment, the improvement suggestions for weak-oracles are generated for every test code location where Reneri detected a program state difference. All the suggestions generated for the same test case and the same extreme transformation are related to the same finding. Reporting all these suggestions is redundant. It is possible to select only one code location by using heuristics as simple as pointing to the first location where the difference is spotted, or the code location closer to an assertion. This improves the output of Reneri.

Reneri instruments the Java source code of a test case. This can avoided to favor bytecode instrumentation and thus facilitating the expansion of the tool to other JVM languages. This instrumentation gathers much more information than what is really needed in the analysis of one method. It is possible to enhance Reneri with a static code inspection to determine the exact method calls and expressions that need to be observed. This would make the dynamic analysis

more efficient. The static analysis that Reneri performs can be done at the bytecode level. This shift imposes then a challenge when reporting the generated suggestions. These reports must comply with the language in which the source code was written.

Embedding the *test intention* in automatic test generation

In this thesis we have implemented a tool that automatically analyses the code and suggests developers what to do to improve their test suites. This approach seems to be, at the moment, more appealing than fully automated test generation tools.

There are several reasons why test generation tools are not adopted by practitioners. Tools like EVOSUITE and DSpot generate tests following adequacy criteria like code coverage or the mutation score. In practice reaching 100% of code coverage or mutation score is not realistic and not even a goal for developers.

By only targeting the mentioned adequacy criteria as fitness functions, these test generation tools leave aside other notions that are relevant for developers. Most of the times the code these tools generate is hard to understand and does not follow good coding practices. The generated test cases are not useful for developers as they are often too trivial, or too complex or, more importantly, do not manifest a clear intent. Test cases without a clear purpose or whose intent is not relevant are dropped and not included in the test suite.

There is no model for the intent of a test case. The concept itself is not well defined. The fitness functions of test generation algorithms do not encode such a notion of purpose.

A way to overcome this challenge could be to initially select concrete and relevant objectives for the test generation process. The intent of the generated test cases would be to address these objectives. For example, the intention of a test case could be to catch an specific undetected extreme transformation discovered by Descartes. This is a concrete intention. The generated test case should reflect this purpose. The relevance of the test case is then determined by the relevance of the method in the code base.

The problem is now transposed to the selection of the most relevant issues from the output of Descartes. The existing test cases can be used as initial solutions for the test generation process. The exploration of the solution space can be guided by the suggestions produced by Reneri. These suggestions point to a direction in which the concrete objective, that is, detecting the extreme transformation can be solved. The move from one solution, a test case, to the other by following these suggestions guarantees an improvement in the fitness function. This assures a faster convergence to a local optimum. This is a viable strategy that can be implemented and studied in future investigations. The success of this alternative is conditioned by the improvement of Descartes and Reneri in the ways we have previously discussed.

To conclude, the long term perspective of this thesis is to close the gap between current test generation techniques and our recent results on generated suggestions for test improvement.

APPENDIX A

LIST OF PROJECTS USED IN CHAPTER 2 AND CHAPTER 3 AND THEIR RESPECTIVE REVISIONS

Legend. Column “Project" list the projects used in chapters 2 and 3. Column “Repository URL" contains links to the available source code. Column “Commit" contains the prefix of the SHA-1 hash identifying the commit with the source code state that was used in this study.

Project	Repository URL	Commit
authzforce	https://github.com/authzforce/core.git	81ae566
aws-sdk-java	https://github.com/aws/aws-sdk-java	b5ae6ce
commons-cli	https://github.com/apache/commons-cli	c246bd4
commons-codec	https://github.com/apache/commons-codec	e9da3d1
commons-collections	https://github.com/apache/commons-collections	db18992
commons-io	https://github.com/apache/commons-io	e36d531
commons-lang	https://github.com/apache/commons-lang	e8f924f
flink-core	https://github.com/apache/flink/tree/master/flink-core	740f711
gson	https://github.com/google/gson	c3d17e3
jaxen	https://github.com/jaxen-xpath/jaxen	a8bd805
jfreechart	https://github.com/jfree/jfreechart	a7156d4
jgit	https://github.com/eclipse/jgit	1513a56
joda-time	https://github.com/JodaOrg/joda-time	6ad1338
jopt-simple	https://github.com/jopt-simple/jopt-simple	b38b70d
jsoup	https://github.com/jhy/jsoup	35e80a7
pdfbox	https://github.com/apache/pdfbox	09e9173
sat4j-core	https://gitlab.ow2.org/sat4j/sat4j/tree/master/org.sat4j.core	1a01276
scifio	https://github.com/scifio/scifio	2760af6
spoon	https://github.com/INRIA/spoon	fd878bc
urbanairship	https://github.com/urbanairship/java-library	aafc049
xwiki-rendering	https://github.com/xwiki/xwiki-rendering	cb3c444

Appendix B

List of projects in Chapter 4 and their respective revisions

Legend. Column "Project" list the projects used in Chapter 4. Column "Repository URL" contains links to the available source code. Column "Revision" contains the tag referencing the commit used in the study.

Project	Repository URL	Revision
spring-petclinic	https://github.com/spring-projects/spring-petclinic	85aab20
jpush-api-java-client	https://github.com/jpush/jpush-api-java-client	v3.3.8
commons-cli	https://github.com/apache/commons-cli	18f8576
jopt-simple	https://github.com/jopt-simple/jopt-simple	31ed6b5
yahoofinance-api	https://github.com/sstrickx/yahoofinance-api	v3.14.0
gson-fire	https://github.com/julman99/gson-fire	de961ba
j2html	https://github.com/tipsy/j2html	cd6e008
eaxy	https://github.com/jhannes/eaxy	5d66430
javapoet	https://github.com/square/javapoet	30a8bda
java-html-sanitizer	https://github.com/OWASP/java-html-sanitizer	release-20180219.1
cron-utils	https://github.com/jmrozanec/cron-utils	6.0.3
commons-codec	https://github.com/apache/commons-codec	5e3b6f7
jsoup	https://github.com/jhy/jsoup	cecdb32
TridentSDK	https://github.com/TridentSDK/TridentSDK	c60da34
jcodemodel	https://github.com/phax/jcodemodel	jcodemodel-3.2.0

APPENDIX C

QUESTIONNAIRE PROVIDED TO DEVELOPERS IN THE ANSWER OF RQ3 IN CHAPTER 4

Questions included on the form describing each testing issue and its corresponding hint. All the options inside a question are exclusive.

Please check the testing issue and hint described in the following URL **https://github.com/...**

- The issue in the description is:
  - relevant
  - of medium relevance
  - not important
  - not really a testing issue
- You would solve the issue by:
  - adding a new assertion to an existing test case (Possibly adding a new method call as well)
  - creating a new test case which is a slight modification of an existing test case
  - creating a completely new test case
  - Other (Specify)
- If you actually solve the issue, please share the URL to the commit
- The suggestion provided in the description
  - points to the exact solution
  - provides helpful information to solve the issue
  - is not really helpful
  - is misleading
- Please, include here any general feedback you would like to provide regarding this issue

Bibliography

[1] Allen Troy Acree Jr., "On Mutation", PhD Thesis, Atlanta, GA, USA: Georgia Institute of Technology, 1980 (cit. on p. 27).

[2] Paul Ammann and Jeff Offutt, *Introduction to software testing*, Cambridge University Press, 2016 (cit. on p. 19).

[3] James H. Andrews, Lionel C. Briand, and Yvan Labiche, "Is Mutation an Appropriate Tool for Testing Experiments?", in: *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, New York, NY, USA: ACM, 2005, pp. 402–411, ISBN: 978-1-58113-963-1, DOI: 10.1145/1062455.1062530, URL: http://doi.acm.org/10.1145/1062455.1062530 (cit. on p. 26).

[4] Kelly Androutsopoulos, David Clark, Haitao Dan, Robert M. Hierons, and Mark Harman, "An analysis of the relationship between conditional entropy and failed error propagation in software testing", in: *Proceedings of the 36th international conference on software engineering*, ACM, 2014, pp. 573–583 (cit. on p. 30).

[5] Taweesup Apiwattanapong, Raul Santelices, Pavan Kumar Chittimalli, Alessandro Orso, and Mary Jean Harrold, "Matrix: Maintenance-oriented testing requirements identifier and examiner", in: *Testing: Academic and Industrial Conference-Practice And Research Techniques, 2006. TAIC PART 2006. Proceedings*, IEEE, 2006, pp. 137–146 (cit. on p. 31).

[6] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon, "Automatic Test Cases Optimization: a Bacteriologic Algorithm", in: *IEEE Software* 22.*2* (Mar. 2005), pp. 76–82, URL: http://www.irisa.fr/triskell/publis/2005/Baudry05d.pdf (cit. on p. 28).

[7] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon, "From Genetic to Bacteriological Algorithms for Mutation-Based Testing", in: *Software, Testing, Verification & Reliability journal (STVR)* 15.*2* (June 2005), pp. 73–96, URL: http://www.irisa.fr/triskell/publis/2005/Baudry05a.pdf (cit. on p. 28).

[8] Michael Braukus, *NASA Honors Apollo Engineer*, Sept. 3, 2003, URL: https://www.nasa.gov/home/hqnews/2003/sep/HQ_03281_Hamilton_Honor.html (visited on 09/27/2019) (cit. on p. 12).

[9] Timothy A. Budd, "Mutation Analysis of Program Test Data", AAI8025191, PhD thesis, New Haven, CT, USA, 1980 (cit. on p. 27).

[10] Pablo C. Cañizares, Alberto Núñez, and Juan de Lara, "OUTRIDER: Optimizing the mUtation Testing pRocess In Distributed EnviRonments", in: *Procedia Computer Science* 108 (2017), International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland, pp. 505–514, ISSN: 1877-0509, DOI: https://doi.org/10.1016/j.procs.2017.05.095, URL: http://www.sciencedirect.com/science/article/pii/S1877050917306440 (cit. on p. 27).

[11] Shigeru Chiba, "Load-Time Structural Reflection in Java", en, in: *ECOOP 2000 — Object-Oriented Programming*, ed. by Elisa Bertino, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2000, pp. 313–336, ISBN: 978-3-540-45102-0 (cit. on p. 75).

[12] Henry Coles, "Mutation Testing: Automate the search for Imperfect Tests", in: *Java Magazine* 6.*6* (Nov. 2016), pp. 43–54, URL: http://www.javamagazine.mozaicreader.com/NovDec2016 (visited on 09/13/2019) (cit. on pp. 24, 31).

[13] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque, "PIT: A Practical Mutation Testing Tool for Java (Demo)", in: *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, New York, NY, USA: ACM, 2016, pp. 449–452, ISBN: 978-1-4503-4390-9, DOI: 10.1145/2931037.2948707, URL: http://doi.acm.org/10.1145/2931037.2948707 (cit. on pp. 24, 30).

[14] Benjamin Danglot, Oscar L. Vera-Pérez, Benoit Baudry, and Martin Monperrus, "Automatic test improvement with DSpot: a study with ten mature open-source projects", en, in: *Empirical Software Engineering* 24.4 (Aug. 2019), pp. 2603–2635, ISSN: 1573-7616, DOI: 10.1007/s10664-019-09692-y, URL: https://doi.org/10.1007/s10664-019-09692-y (cit. on pp. 3, 16, 42, 72, 88).

[15] Benjamin Danglot, Oscar L. Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus, and Benoit Baudry, "A snowballing literature study on test amplification", in: *Journal of Systems and Software* 157 (Nov. 2019), p. 110398, ISSN: 0164-1212, DOI: 10.1016/j.jss.2019.110398, URL: http://www.sciencedirect.com/science/article/pii/S0164121219301736 (cit. on pp. 3, 16).

[16] Brett Daniel, Vilas Jagannath, Danny Dig, and Darko Marinov, "ReAssert: Suggesting Repairs for Broken Unit Tests", in: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, Washington, DC, USA: IEEE Computer Society, 2009, pp. 433–444, ISBN: 978-0-7695-3891-4, DOI: 10.1109/ASE.2009.17, URL: https://doi.org/10.1109/ASE.2009.17 (visited on 02/14/2019) (cit. on pp. 29, 32, 99).

[17] Murial Daran and Pascale Thévenod-Fosse, "Software Error Analysis: A Real Case Study Involving Real Faults and Mutations", in: *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '96, New York, NY, USA: ACM, 1996, pp. 158–171, ISBN: 978-0-89791-787-2, DOI: 10.1145/229000.226313, URL: http://doi.acm.org/10.1145/229000.226313 (cit. on p. 26).

[18] Mickaël Delahaye and Lydie du Bousquet, "A Comparison of Mutation Analysis Tools for Java", in: *2013 13th International Conference on Quality Software*, July 2013, pp. 187–195, DOI: 10.1109/QSIC.2013.47 (cit. on p. 31).

[19] Marcio E. Delamaro, Jeff Offutt, and Paul Ammann, "Designing Deletion Mutation Operators", in: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, Mar. 2014, pp. 11–20, DOI: 10.1109/ICST.2014.12 (cit. on p. 27).

[20] Julien Delplanque, Stéphane Ducasse, Guillermo Polito, Andrew P. Black, and Anne Etien, "Rotten Green Tests", in: *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, Montreal, Quebec, Canada: IEEE Press, 2019, pp. 500–511, DOI: 10.1109/ICSE.2019.00062, URL: https://doi.org/10.1109/ICSE.2019.00062 (cit. on p. 32).

[21] Richard A. DeMillo, Edward W. Krauser, and Aditya P. Mathur, "Compiler-integrated program mutation", in: *[1991] Proceedings The Fifteenth Annual International Computer Software Applications Conference*, Sept. 1991, pp. 351–356, DOI: 10.1109/CMPSAC.1991.170202 (cit. on p. 27).

[22] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward, "Hints on test data selection: Help for the practicing programmer", in: *Computer* 11.4 (1978), pp. 34–41 (cit. on pp. 5, 13, 22, 26, 52).

[23] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward, "Program mutation: A new approach to program testing", in: *Infotech State of the Art Report, Software Testing* 2.*1979* (1979), pp. 107–126 (cit. on p. 22).

[24] Richard A. DeMillo and A. Jefferson Offutt, "Constraint-based automatic test data generation", in: *IEEE Transactions on Software Engineering* 17.9 (Sept. 1991), pp. 900–910, ISSN: 0098-5589, DOI: 10.1109/32.92910 (cit. on pp. 25, 28, 30, 65).

[25] Lin Deng, Jeff Offutt, and Nan Li, "Empirical Evaluation of the Statement Deletion Mutation Operator", in: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, Mar. 2013, pp. 84–93, DOI: 10.1109/ICST.2013.20 (cit. on p. 27).

[26] Vinicius H. S. Durelli, Nilton M. De Souza, and Marcio E. Delamaro, "Are Deletion Mutants Easier to Identify Manually?", in: *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Mar. 2017, pp. 149–158, DOI: 10.1109/ICSTW.2017.32 (cit. on pp. 24, 27, 63).

[27] Gordon Fraser and Andrea Arcuri, "Achieving scalable mutation-based generation of whole test suites", en, in: *Empirical Software Engineering* 20.3 (June 2015), pp. 783–812, ISSN: 1382-3256, 1573-7616, DOI: 10.1007/s10664-013-9299-z (cit. on pp. 28, 88).

[28] Gordon Fraser and Andreas Zeller, "Generating parameterized unit tests", in: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ACM, 2011, pp. 364–374 (cit. on p. 29).

[29] Gordon Fraser and Andreas Zeller, "Mutation-Driven Generation of Unit Tests and Oracles", in: *IEEE Transactions on Software Engineering* 38.*2* (Mar. 2012), pp. 278–292, ISSN: 0098-5589, DOI: 10.1109/TSE.2011.93 (cit. on p. 28).

[30] Rahul Gopinath, Iftekhar Ahmed, Mohammad Amin Alipour, Carlos Jensen, and Alex Groce, "Does choice of mutation tool matter?", en, in: *Software Quality Journal* 25.*3* (Sept. 2017), pp. 871–920, ISSN: 0963-9314, 1573-1367, DOI: 10.1007/s11219-016-9317-7, URL: http://link.springer.com/10.1007/s11219-016-9317-7 (visited on 08/02/2017) (cit. on p. 31).

[31] Rahul Gopinath, Carlos Jensen, and Alex Groce, "Code Coverage for Suite Evaluation by Developers", in: *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, Hyderabad, India: ACM, 2014, pp. 72–82, ISBN: 978-1-4503-2756-5, DOI: 10.1145/2568225.2568278, URL: http://doi.acm.org/10.1145/2568225.2568278 (cit. on pp. 20, 22).

[32] Rahul Gopinath, Carlos Jensen, and Alex Groce, "Mutations: How close are they to real faults?", in: *Software reliability engineering (ISSRE), 2014 IEEE 25th international symposium on*, IEEE, 2014, pp. 189–200, URL: http://ieeexplore.ieee.org/abstract/document/6982626/ (visited on 08/02/2017) (cit. on p. 26).

[33] Rahul Gopinath, Carlos Jensen, and Alex Groce, "The Theory of Composite Faults", in: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Mar. 2017, pp. 47–57, DOI: 10.1109/ICST.2017.12 (cit. on p. 26).

[34] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand, "Experiments of the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria", in: *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, Sorrento, Italy: IEEE Computer Society Press, 1994, pp. 191–200, ISBN: 0-8186-5855-X, URL: http://dl.acm.org/citation.cfm?id=257734.257766 (cit. on p. 32).

[35] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella, "Test oracle assessment and improvement", in: *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ACM, 2016, pp. 247–258 (cit. on p. 29).

[36] *JUnit project home page*, URL: https://junit.org/ (visited on 09/25/2019) (cit. on p. 19).

[37] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser, "Are Mutants a Valid Substitute for Real Faults in Software Testing?", in: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, New York, NY, USA: ACM, 2014, pp. 654–665, ISBN: 978-1-4503-3056-5, DOI: 10.1145/2635868.2635929, URL: http://doi.acm.org/10.1145/2635868.2635929 (visited on 03/30/2017) (cit. on p. 26).

[38] René Just, Franz Schweiggert, and Gregory M. Kapfhammer, "MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler", in: *Proceedings of the International Conference on Automated Software Engineering (ASE)*, Nov. 9, 2011, pp. 612–615 (cit. on pp. 30, 99).

[39] Vigdis By Kampenes, Tore Dybå, Jo Erskine Hannay, and Dag I. K. Sjøberg, "A systematic review of effect size in software engineering experiments", in: *Information & Software Technology* 49.*11-12* (2007), pp. 1073–1086, DOI: 10.1016/j.infsof.2007.02.015 (cit. on p. 53).

[40] Marinos Kintis, Mike Papadakis, and Nicos Malevris, "Employing second-order mutation for isolating first-order equivalent mutants", en, in: *Software Testing, Verification and Reliability* 25.*5-7* (Aug. 2015), pp. 508–535, ISSN: 1099-1689, DOI: 10.1002/stvr.1529, URL: http://onlinelibrary.wiley.com/doi/10.1002/stvr.1529/abstract (visited on 03/30/2017) (cit. on p. 27).

[41] Marinos Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, and Nicos Malevris, "Analysing and Comparing the Effectiveness of Mutation Testing Tools: A Manual Study", in: *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Oct. 2016, pp. 147–156, DOI: 10.1109/SCAM.2016.28 (cit. on p. 31).

[42] Bob Kurtz, Paul Ammann, Jeff Offutt, Márcio E Delamaro, Mariet Kurtz, and Nida Gökçe, “Analyzing the validity of selective mutation with dominator mutants”, en, in: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, Seattle, WA, USA: ACM Press, 2016, pp. 571–582, ISBN: 978-1-4503-4218-6, DOI: 10.1145/2950290.2950322, URL: http://dl.acm.org/citation.cfm?doid=2950290.2950322 (visited on 08/27/2018) (cit. on p. 27).

[43] Bob Kurtz, Paul Ammann, Jeff Offutt, and Mariet Kurtz, “Are We There Yet? How Redundant and Equivalent Mutants Affect Determination of Test Completeness”, in: *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Apr. 2016, pp. 142–151, DOI: 10.1109/ICSTW.2016.41 (cit. on pp. 27, 98).

[44] Thomas Laurent, Mike Papadakis, Marinos Kintis, Christopher Henard, Yves Le Traon, and Anthony Ventresque, “Assessing and Improving the Mutation Testing Practice of PIT”, in: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Mar. 2017, pp. 430–435, DOI: 10.1109/ICST.2017.47 (cit. on p. 31).

[45] Nan Li and Jeff Offutt, “Test Oracle Strategies for Model-Based Testing”, in: *IEEE Transactions on Software Engineering* 43.*4* (Apr. 2017), pp. 372–395, ISSN: 0098-5589, DOI: 10.1109/TSE.2016.2597136 (cit. on pp. 25, 30).

[46] Huan Lin, Yawen Wang, Yunzhan Gong, and Dahai Jin, “Domain-RIP Analysis: A Technique for Analyzing Mutation Stubbornness”, in: *IEEE Access* 7 (2019), pp. 4006–4023, ISSN: 2169-3536, DOI: 10.1109/ACCESS.2018.2883776 (cit. on p. 30).

[47] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Józala, “Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation”, in: *IEEE Transactions on Software Engineering* 40.*1* (Jan. 2014), pp. 23–42, ISSN: 0098-5589, DOI: 10.1109/TSE.2013.44 (cit. on p. 24).

[48] Larry J. Morell, *A Theory of Error-Based Testing.* en, tech. rep. TR-1395, MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE, Apr. 1984, URL: https://apps.dtic.mil/docs/citations/ADA143533 (visited on 07/22/2019) (cit. on pp. 25, 30).

[49] Larry J. Morell, “A theory of fault-based testing”, in: *IEEE Transactions on Software Engineering* 16.*8* (Aug. 1990), pp. 844–857, ISSN: 0098-5589, DOI: 10.1109/32.57623 (cit. on pp. 25, 30, 65).

[50] Jakub Možucha and Bruno Rossi, “Is Mutation Testing Ready to Be Adopted Industry-Wide?”, en, in: *Product-Focused Software Process Improvement*, Lecture Notes in Computer Science, Springer, Cham, Nov. 2016, pp. 217–232, ISBN: 978-3-319-49093-9, DOI: 10.1007/978-3-319-49094-6_14, URL: https://link.springer.com/chapter/10.1007/978-3-319-49094-6_14 (visited on 07/27/2017) (cit. on p. 24).

[51] Rainer Niedermayr, Elmar Juergens, and Stefan Wagner, “Will my tests tell me if I break this code?”, en, in: *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*, New York, NY, USA: ACM Press, 2016, pp. 23–29, ISBN: 978-1-4503-4157-8, DOI: 10.1145/2896941.2896944, URL: http://dl.acm.org/citation.cfm?doid=2896941.2896944 (visited on 01/05/2017) (cit. on pp. 5, 7, 13, 22, 25, 26, 27, 33, 37, 39, 46, 47, 48, 49, 50, 52, 63).

[52] Rainer Niedermayr and Stefan Wagner, “Is the Stack Distance Between Test Case and Method Correlated With Test Effectiveness?”, in: *Proceedings of the Evaluation and Assessment on Software Engineering*, EASE '19, event-place: Copenhagen, Denmark, New York, NY, USA: ACM, 2019, pp. 189–198, ISBN: 978-1-4503-7145-2, DOI: 10.1145/3319008.3319021, URL: http://doi.acm.org/10.1145/3319008.3319021 (visited on 08/02/2019) (cit. on pp. 25, 28, 42, 95).

[53] A. Jefferson Offutt, Gregg Rothermel, and Christian Zapf, “An Experimental Evaluation of Selective Mutation”, in: *Proceedings of the 15th International Conference on Software Engineering*, ICSE '93, Los Alamitos, CA, USA: IEEE Computer Society Press, 1993, pp. 100–107, ISBN: 978-0-89791-588-5, URL: http://dl.acm.org/citation.cfm?id=257572.257597 (visited on 08/30/2018) (cit. on p. 27).

[54] Carlos Pacheco and Michael D. Ernst, “Eclat: Automatic Generation and Classification of Test Inputs”, en, in: *ECOOP 2005 - Object-Oriented Programming*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, July 2005, pp. 504–527, ISBN: 978-3-540-27992-1, DOI: 10.1007/11531142_22, URL: https://link.springer.com/chapter/10.1007/11531142_22 (visited on 07/06/2018) (cit. on p. 29).

[55] Carlos Pacheco and Michael D. Ernst, “Randoop: Feedback-directed Random Testing for Java”, in: *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA ’07, New York, NY, USA: ACM, 2007, pp. 815–816, ISBN: 978-1-59593-865-7, DOI: 10.1145/1297846.1297902 (cit. on p. 28).

[56] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman, “Chapter Six - Mutation Testing Advances: An Analysis and Survey”, in: ed. by Atif M. Memon, vol. 112, Advances in Computers, Elsevier, 2019, pp. 275–378, DOI: 10.1016/bs.adcom.2018.03.015, URL: http://www.sciencedirect.com/science/article/pii/S0065245818300305 (cit. on pp. 27, 30).

[57] Ali Parsai, Alessandro Murgia, and Serge Demeyer, “LittleDarwin: A Feature-Rich and Extensible Mutation Testing Framework for Large and Complex Java Systems”, in: *Fundamentals of Software Engineering*, ed. by Mehdi Dastani and Marjan Sirjani, Cham: Springer International Publishing, 2017, pp. 148–163, ISBN: 978-3-319-68972-2 (cit. on p. 30).

[58] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier, “SPOON: A library for implementing analyses and transformations of Java source code”, in: *Softw., Pract. Exper.* 46 (2016), pp. 1155–1179, DOI: 10.1002/spe.2346 (cit. on p. 75).

[59] Goran Petrović and Marko Ivanković, “State of Mutation Testing at Google”, in: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP ’18, Gothenburg, Sweden: ACM, 2018, pp. 163–171, ISBN: 978-1-4503-5659-6, DOI: 10.1145/3183519.3183521, URL: http://doi.acm.org/10.1145/3183519.3183521 (cit. on pp. 26, 31, 62).

[60] Goran Petrović, Marko Ivanković, Bob Kurtz, Paul Ammann, and René Just, “An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions”, in: *Proceedings of the International Workshop on Mutation Analysis (Mutation)*, Apr. 2018, pp. 47–53 (cit. on pp. 31, 34, 98).

[61] Alessandro Viola Pizzoleto, Fabiano Cutigi Ferrari, Jeff Offutt, Leo Fernandes, and Márcio Ribeiro, “A systematic literature review of techniques and metrics to reduce the cost of mutation testing”, in: *Journal of Systems and Software* 157 (Nov. 2019), p. 110388, ISSN: 0164-1212, DOI: 10.1016/j.jss.2019.07.100, URL: http://www.sciencedirect.com/science/article/pii/S0164121219301554 (visited on 09/16/2019) (cit. on p. 27).

[62] Macario Polo, Mario Piattini, and Ignacio García-Rodríguez, “Decreasing the cost of mutation testing with second-order mutants”, en, in: *Software Testing, Verification and Reliability* 19.*2* (June 2009), pp. 111–131, ISSN: 09600833, 10991689, DOI: 10.1002/stvr.392, URL: http://doi.wiley.com/10.1002/stvr.392 (visited on 03/31/2017) (cit. on p. 27).

[63] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri, “Seeding strategies in search-based unit test generation”, in: *Software Testing, Verification and Reliability* 26.*5* (2016), pp. 366–401 (cit. on p. 28).

[64] Iman Saleh and Khaled Nagi, “HadoopMutator: A Cloud-Based Mutation Testing Framework”, en, in: *Software Reuse for Dynamic Systems in the Cloud and Beyond*, ed. by Ina Schaefer and Ioannis Stamelos, Lecture Notes in Computer Science, Springer International Publishing, 2014, pp. 172–187, ISBN: 978-3-319-14130-5 (cit. on p. 27).

[65] Raul Santelices, Pavan Kumar Chittimalli, Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold, “Test-suite augmentation for evolving software”, in: *23rd IEEE/ACM International Conference on*, IEEE, 2008, pp. 218–227, URL: https://pdfs.semanticscholar.org/05cf/2988ea3ac5e697fc51f85e7dd2031dd8af01.pdf (cit. on p. 32).

[66] Raul Santelices and Mary Jean Harrold, “Applying aggressive propagation-based strategies for testing changes”, in: *IEEE Fourth International Conference on Software Testing, Verification and Validation*, IEEE, 2011, pp. 11–20 (cit. on p. 32).

[67] David Schuler and Andreas Zeller, “Checked coverage: an indicator for oracle quality”, in: *Software Testing, Verification and Reliability* 23.*7* (2013), pp. 531–551 (cit. on p. 29).

[68] Forrest J. Shull, Jeffrey C. Carver, Sira Vegas, and Natalia Juristo, “The role of replications in Empirical Software Engineering”, en, in: *Empirical Software Engineering* 13.*2* (Apr. 2008), pp. 211–218, ISSN: 1382-3256, 1573-7616, DOI: 10.1007/s10664-008-9060-1, URL: http://link.springer.com/10.1007/s10664-008-9060-1 (visited on 02/23/2018) (cit. on p. 45).

[69] Ben H. Smith and Laurie Williams, “On guiding the augmentation of an automated test suite via mutation analysis”, in: *Empirical Software Engineering* 14.*3* (2009), pp. 341–369, URL: https://repository.lib.ncsu.edu/bitstream/handle/1840.4/1967/TR-2008-9.pdf?sequence=1 (cit. on p. 28).

[70] Ben H. Smith and Laurie Williams, “Should software testers use mutation analysis to augment a test set?”, in: *Journal of Systems and Software*, SI: TAIC PART 2007 and MUTATION 2007 82.*11* (Nov. 2009), pp. 1819–1832, ISSN: 0164-1212, DOI: 10.1016/j.jss.2009.06.031, URL: http://www.sciencedirect.com/science/article/pii/S0164121209001368 (visited on 07/03/2018) (cit. on p. 28).

[71] Yoonki Song, Suresh Thummalapenta, and Tao Xie, “UnitPlus: Assisting Developer Testing in Eclipse”, in: *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse ’07, event-place: Montreal, Quebec, Canada, New York, NY, USA: ACM, 2007, pp. 26–30, ISBN: 978-1-60558-015-9, DOI: 10.1145/1328279.1328285, URL: http://doi.acm.org/10.1145/1328279.1328285 (visited on 08/06/2019) (cit. on p. 32).

[72] Matt Staats, Gregory Gay, and Mats P. E. Heimdahl, “Automated oracle creation support, or: How I learned to stop worrying about fault propagation and love mutation testing”, in: *2012 34th International Conference on Software Engineering (ICSE)*, June 2012, pp. 870–880, DOI: 10.1109/ICSE.2012.6227132 (cit. on p. 29).

[73] Mustafa M Tikir and Jeffrey K. Hollingsworth, “Efficient instrumentation for code coverage testing”, in: *ACM SIGSOFT Software Engineering Notes*, vol. 27, 4, ACM, 2002, pp. 86–96 (cit. on p. 20).

[74] Nicholas Tocci and Gregory M. Kapfhammer, *Automatic Detection of Pseudo-tested Methods using Python and Pytest*, Pycon 2019, 2019, URL: https://us.pycon.org/2019/schedule/presentation/112/#! (visited on 09/29/2019) (cit. on p. 98).

[75] Roland H. Untch, “On Reduced Neighborhood Mutation Analysis Using a Single Mutagenic Operator”, in: *Proceedings of the 47th Annual Southeast Regional Conference*, ACM-SE 47, Clemson, South Carolina: ACM, 2009, 71:1–71:4, ISBN: 978-1-60558-421-8, DOI: 10.1145/1566445.1566540, URL: http://doi.acm.org/10.1145/1566445.1566540 (cit. on pp. 27, 99).

[76] Simon Urli, Zhongxing Yu, Lionel Seinturier, and Martin Monperrus, “How to Design a Program Repair Bot?: Insights from the Repairnator Project”, in: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP ’18, Gothenburg, Sweden: ACM, 2018, pp. 95–104, ISBN: 978-1-4503-5659-6, DOI: 10.1145/3183519.3183540, URL: http://doi.acm.org/10.1145/3183519.3183540 (cit. on p. 77).

[77] Pascal Urso, Pedro Velho, Mael Audren, Jesús Gorroñogoitia, Fernando Mendez, Lars Boye, Kenneth Skaar, Vincent Massol, Assad Montasser, and Cédric Thomas, *Use Cases Validation Report. STAMP Deliverable 5.6*, Nov. 6, 2018, URL: https://github.com/STAMP-project/docs-forum/blob/f0da6b04e31fe4a964b3e5f62fdfff3a7ef88389/docs/d56_uc_validation_report.pdf (visited on 09/25/2019) (cit. on p. 24).

[78] Oscar Luis Vera-Pérez, Benjamin Danglot, Martin Monperrus, and Benoit Baudry, “A Comprehensive Study of Pseudo-tested Methods”, in: *Empirical Software Engineering* 24.*3* (June 2019), pp. 1195–1225, DOI: 10.1007/s10664-018-9653-2, URL: https://doi.org/10.1007/s10664-018-9653-2 (cit. on pp. 3, 15).

[79] Oscar Luis Vera-Pérez, Benjamin Danglot, Martin Monperrus, and Benoit Baudry, *Suggestions on Test Suite Improvements with Automatic Infection and Propagation Analysis*, Submitted to Transactions in Software Engineering, 2019, arXiv: 1909.04770 [cs.SE] (cit. on pp. 3, 15).

[80] Oscar Luis Vera-Pérez, Martin Monperrus, and Benoit Baudry, "Descartes: A PITest Engine to Detect Pseudo-Tested Methods", in: *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*, 2018, pp. 908–911, DOI: 10.1145/3238147.3240474, URL: https://dl.acm.org/citation.cfm?doid=3238147.3240474 (cit. on pp. 3, 15, 75).

[81] Jeffrey M. Voas, "PIE: a dynamic failure-based technique", in: *IEEE Transactions on Software Engineering* 18.*8* (Aug. 1992), pp. 717–727, ISSN: 0098-5589, DOI: 10.1109/32.153381 (cit. on pp. 25, 30, 65, 99).

[82] Jeffrey M. Voas, "Software testability measurement for intelligent assertion placement", en, in: *Software Quality Journal* 6.*4* (Dec. 1997), pp. 327–336, ISSN: 0963-9314, 1573-1367, DOI: 10.1023/A:1018532607070, URL: http://link.springer.com/article/10.1023/A:1018532607070 (visited on 02/08/2017) (cit. on p. 30).

[83] Jeffrey M. Voas and Keith. W. Miller, "Software testability: the new verification", in: *IEEE Software* 12.*3* (May 1995), pp. 17–28, ISSN: 0740-7459, DOI: 10.1109/52.382180 (cit. on p. 30).

[84] W. Eric Wong and Aditya P. Mathur, "Reducing the cost of mutation testing: An empirical study", en, in: *Journal of Systems and Software* 31.*3* (Dec. 1995), pp. 185–196, ISSN: 0164-1212, DOI: 10.1016/0164-1212(94)00098-0, URL: https://www.sciencedirect.com/science/article/pii/0164121294000980 (visited on 08/30/2018) (cit. on p. 27).

[85] Weichen Eric Wong, "On Mutation and Data Flow", PhD Thesis, West Lafayette, IN, USA: Purdue University, 1993 (cit. on p. 27).

[86] Tao Xie, "Augmenting Automatically Generated Unit-Test Suites with Regression Oracle Checking", in: *ECOOP 2006 – Object-Oriented Programming*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, July 2006, pp. 380–403, ISBN: 978-3-540-35726-1, DOI: 10.1007/11785477_23 (cit. on p. 29).

[87] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining", in: *Empirical Software Engineering* 16.*3* (2011), pp. 325–364 (cit. on p. 19).

[88] Lingming Zhang, Milos Gligoric, Darko Marinov, and Sarfraz Khurshid, "Operator-based and random mutant selection: Better together", in: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2013, pp. 92–102, DOI: 10.1109/ASE.2013.6693070 (cit. on p. 27).

[89] Lu Zhang, Shan-Shan Hou, Jun-Jue Hu, Tao Xie, and Hong Mei, "Is Operator-based Mutant Selection Superior to Random Mutant Selection?", in: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, New York, NY, USA: ACM, 2010, pp. 435–444, ISBN: 978-1-60558-719-6, DOI: 10.1145/1806799.1806863, URL: http://doi.acm.org/10.1145/1806799.1806863 (visited on 08/30/2018) (cit. on p. 27).

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre: Analyse dynamique du programme pour suggérer des améliorations de test aux développeurs

Mot clés : Test de logiciels – analyse de programmes – transformations extrêmes – amélioration de tests – développeurs de logiciels

Resumé : Les tests automatisés sont au cœur du développement de logiciels modernes. Pourtant, les développeurs éprouvent des difficultés lorsqu'il s'agit d'évaluer la qualité de leurs scénarios de test et de trouver des moyens de les améliorer. Le but principal de cette thèse est précisément de générer des suggestions concrètes que les développeurs peuvent suivre pour améliorer leurs suites de tests. Nous proposons l'utilisation de mutations extrêmes ou de transformations extrêmes comme alternative pour découvrir les problèmes de test. Les transformations extrêmes sont une forme de test par mutation qui supprime toute la logique d'une méthode au lieu d'effectuer un petit changement syntaxique dans le code. Comme son homologue traditionnel, elle challenge la suite de tests avec une variante du programme pour voir si les cas de tests peuvent détecter le changement. Dans cette thèse, nous évaluons la pertinence des problèmes de test que les transformations extrêmes peuvent mettre en évidence. Nous proposons également une analyse dynamique de infection-propagation pour dériver automatiquement des suggestions concrètes d'amélioration des tests à partir de transformations extrêmes non détectées. Nos résultats sont validés par les échanges avec les développeurs. Nous faisons également état de l'adoption industrielle de certaines parties de nos résultats.

Title: Dynamic program analysis for suggesting test improvements to developers

Keywords : Software testing – program analysis – extreme transformations – test improvements – software developers

Abstract : Automated testing is at the core of modern software development. Yet developers struggle when it comes to the evaluation of the quality of their test cases and how to improve them. The main goal of this thesis is precisely that, to generate concrete suggestion that developers can follow to improve their test suite. We propose the use of extreme mutation, or extreme transformations as an alternative to discover testing issues. Extreme transformations are a form of mutation testing that remove the entire logic of a method instead of making a small syntactic change in the code. As it traditional counterpart it challenges the test suite with a transformed variant of the program to see if the test cases can detect the change. In this thesis we assess the relevance of the testing issues that extreme transformations can spot. We also propose a dynamic infection-propagation analysis to automatically derive concrete test improvement suggestions from undetected extreme transformations. Our results are validated through the interaction with actual developers. We also report the industrial adoption of parts of our results. developers to improve their tests by detecting more of these transformations. Our results are validated through the interaction with actual developers.