MASTER EN SCIENCES INFORMATIQUES

Revue de la littérature sur le débogage de flaky test

Delvaux, Gaetan

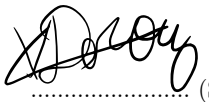
Award date:
2023

Awarding institution:
Universite de Namur

UNIVERSITÉ DE NAMUR
Faculté d'informatique
Année académique 2022-2023

Revue de la littérature sur le débogage de flaky test

Delvaux Gaëtan


..... (Signature pour approbation du dépôt - REE art. 40)

Promoteur : Devroey X.

Mémoire présenté en vue de l'obtention du grade de Master 60 en Sciences Informatiques

Faculté d'Informatique – Université de Namur

RUE GRANDGAGNAGE, 21 ● B-5000 NAMUR(BELGIUM)

Remerciements

La réalisation de ce mémoire a été possible grâce au soutien inconditionnel de plusieurs personnes à qui je voudrais témoigner toute ma gratitude.

Je voudrais tout d'abord adresser toute ma reconnaissance à mon promoteur, Monsieur Devroey Xavier, pour sa patience, ses conseils et son encouragement durant mes moments de doutes.

Je désire aussi remercier ASSAR et son équipe IT, pour leur confiance et leur soutien.

Mes remerciements s'adressent également à ma famille et mes proches, pour leur appui et leur encouragement.

Enfin, je tiens à remercier spécialement Wannez Adeline, pour son courage et sa force de caractère.

Résumé

Les flaky tests sont des tests non déterministes, ils peuvent donner des résultats différents sans modification du code. Ce qui entraîne une perte de temps et de ressources. Une meilleure connaissance de ce domaine doit mener à une diminution de ces désagréments. Cependant, il y a peu de travaux faisant l'effort de regrouper ces connaissances.

C'est pourquoi, ce mémoire va proposer de répondre à ces questions "*Quelles sont les catégories de flaky tests identifiées ?*" et "*Quelles techniques de debug sont les plus appropriées pour les différentes catégories de flaky tests ?*".

Pour ce faire, une mapping study ainsi qu'une revue systématique de la littérature ont été mises en place. Ce qui permettra de faire le point sur les connaissances actuelles. De présenter les différentes catégories proposées ainsi que leurs classifications. Sans oublier de discuter des besoins et envies des développeurs pour le futur.

Nous verrons aussi que la recherche ne doit pas s'arrêter là, car il n'y pas encore de proposition forte pour lier les catégories aux techniques de debug. Des travaux de recherches sont encore nécessaires pour arriver à proposer des méthodes regroupant l'ensemble des outils et techniques proposés.

Mots-clés *: Flaky test, Revue de la littérature systématique, Mapping study, Classification, Détection, Reproduction, Vue des développeurs*

Abstract

Flaky tests are nondeterministic tests, they can give different results without changes to the code. This wastes time and resources. A better understanding of this field should lead to a decrease of these inconveniences. However, there is little work that makes the effort to bring this knowledge together.

This is why this thesis will propose to answer these questions "*What are the categories of flaky tests identified?*" and "*What debugging techniques are the most appropriate for the different categories of flaky tests?"*

To do this, a mapping study and a systematic review of the literature have been set up. This will allow us to take stock of current knowledge. To present the different categories proposed as well as their classifications. Without forgetting to discuss the needs and desires of developers for the future.

We will also see that the research should not stop there, as there is not yet a strong proposal to link the categories to debugging techniques. Research work is still needed to come up with methods that bring together all the proposed tools and techniques.

Keywords *: Flaky test, Systematic Literatur Review, Mapping study, Classification, Detection, Replication, Developer's view*

Contents

Acronymes

ADINS Assumes a Deterministic Implementation of a Non-deterministic Specification. 36, 37

API Application Programming Interface. 33, 42, 43

APR Automated Program Repair. 33

AST Abstract-syntax tree. 35, 36

CI Continuous integration. 11, 12, 56

FIT Flakiness-inducing test. 33

FITTER Flakiness inducing test creation and repair. 33, 34, 38

MLR Multivocal Literatur Review. 15

NOD Non-order-dependent. 34, 35

OD Order-dependent. 34, 35

RTS Regression Test Selection. 11, 56, 57

SLR Systematic Literature Review. 10, 14–16, 22, 25

UI User Interface. 53, 54, 56

VCS Version Control System. 35

Glossaire

core dump : Fichier dans lequel est enregistré une copie de la mémoire vive et des registres d'un processeur. 48

EVOSUITE : C'est un framework[15] qui applique un algorithme génétique pour produire une suite de tests pour les logiciels orienté objet. 48

stack trace : Représente la pile d'exécution à un moment donné lors de l'exécution d'un programme informatique. 46–52

1 Introduction

Dans la vie d'un développeur, créer un logiciel répondant à ses spécifications et aux besoins du client est une priorité. Pour cela, ceux-ci mettent en place deux types de tests, les tests de validation qui permettront de vérifier que le logiciel répond effectivement aux attentes du client et les tests de vérification qui devront prévenir le développeur si une de ses modifications rompt une spécification du logiciel. Ces tests de vérification, aussi appelés tests de régression, doivent être menés de manière itérative.

Depuis quelques années, les chercheurs se sont rendus compte que certains tests de régression étaient non-déterministes, c'est à dire qu'un test non modifié ne renvoie pas toujours la même réponse alors qu'ils sont sensés le faire. Ces tests non déterministes sont appelés des "flaky tests".

Les flaky tests sont un problème important dans la vie d'un développeur. Ils entraînent une perte de temps dûe à la recherche de ce qui a apporté l'indéterminisme, une perte de confiance envers le système de tests dans son ensemble et le risque de laisser passer des erreurs.

En plus de cela, la gestion de ces tests représente un coût très important pour l'entreprise. La technique principale pour les faire survenir étant la ré-exécution de l'ensemble des tests, cela demande énormément de ressources et de temps. C'est pourquoi, de nombreux chercheurs ont proposé des études sur le domaine.

Ce travail a pour but de faire le tour du dit domaine et de connaître l'état de la recherche sur les flaky tests et les différentes méthodes de gestion existantes. Pour arriver à cela, nous nous sommes aidé d'une question permettant de réduire l'ampleur du travail et de lui donner une direction.

"*Quelles sont les catégories de flaky tests identifiées ?*" et sur "*Quelles techniques de debug sont les plus appropriées pour les différentes catégories de flaky tests ?*".

Pour tenter de répondre à ces questions, ce travail va utiliser différentes techniques de recherche: une mapping study sera suivie pour arriver à récolter les articles traitant du sujet. Elle nous permettra de fournir des données factuelles pour étayer nos propos. Une revue systématique de la littérature suivra et sera employée pour proposer des réponses aux questions précédemment citées.

Ce travail présentera diverses sections faisant le lien avec les flaky tests et ayant chacune pour objectif de synthétiser l'état des connaissances et d'apporter un angle d'approche différent pour explorer la problématique dans son ensemble. La vision des développeurs, les méthodes de détection et de prédiction, la classification et la reproduction d'erreurs sont autant de chemins menant à la compréhension du domaine.

Pour finir, nous pourrons examiner les données obtenues et les connaissances partagées. Ce sera l'occasion de faire le point sur la situation et de proposer des pistes pour les recherches futures.

2 État de l'art

L'état de l'art reprend l'état des connaissances sur un domaine spécifique à une date précise. Dans notre cas, il a été mené pour différents domaines qui sont nécessaires à la bonne compréhension de ce travail. La méthodologie de recherche, les bases du test logiciel, la présentation du concept de débogage et la présentation du concept de flaky test sont les domaines sélectionnés pour permettre au lecteur d'avoir une base suffisante à la bonne compréhension de ce travail.

Cette partie est importante, car elle permet de mettre en contexte l'ensemble du travail qui suit et donne les clefs de compréhension pour des lecteurs moins avertis. Comme pour tous travaux de recherche, les idées ne seront pas toujours développées à leurs paroxysmes, cependant, des articles aidant à approfondir l'idée sont proposés pour diriger le lecteur vers des documents pouvant répondre à ses interrogations. Cette méthode est appelée snowball et lui permettra d'élargir ses connaissances sur les sujets qui l'intéressent ou le questionnent.

2.1 Revue systématique de la littérature

Une approche systématique de la littérature, ou bien revue systématique, est une méthodologie proposée, pour le domaine logiciel, par Kitchenham & al.[21] Celle-ci vise à collecter, évaluer et interpréter de manière critique toutes les recherches disponibles en rapport avec un thème spécifique en utilisant un processus systématique. Ce travail demande à une personne d'agir de façon scientifique, c'est à dire, de présenter une évaluation du sujet en utilisant une méthodologie fiable, rigoureuse et vérifiable en minimisant les biais pouvant introduire des idées ou des réponses erronées.

L'intérêt d'utiliser cette méthode se situe dans sa reproductibilité par une autre personne. En suivant cette méthodologie, toute personne devrait arriver au même résultat que la personne ayant fait la revue systématique.

Cette méthodologie peut, par exemple, suivre les étapes définies par Kitchenham & al.[21] :

1. Planifier l'étude - identifier le problème qui est posé et qui doit servir à définir la question de recherche.
   (a) Identifier le but de l'étude.
   (b) Spécifier la question de recherche.
   (c) Développer le protocole de la revue.
      - Utilisation des articles identifiés par un expert du domaine.
      - Utilisation de mots clefs.
      - Technique du 'snowball' à la lecture des articles.

- Utilisation de valeurs de recherche.
- Utilisation de critères d'inclusion et d'exclusion.

2. Faire l'étude.
   (a) Identifier et sélectionner les articles en lien avec la recherche.
   (b) Évaluer les articles sélectionnés.
   (c) Extraire et synthétiser les données.
3. Présentation de l'étude.

Dans ce mémoire, la méthodologie de Kitchenham & al. a été suivie. Ce choix a été fait pour apporter une structure connue et éprouvée au processus utilisé et ainsi améliorer la qualité de l'étude.

2.2 Mapping study

Une mapping study est une méthode servant à construire un schéma de classification et à structurer un domaine de recherche. Elle doit donner une vue d'ensemble du domaine et identifier la quantité et le type de recherches disponibles pour celui-ci. Pour faire ce travail, il a été décidé de prendre comme référence, pour la mapping study, le travail présenté par Petersen & al.[31] et la présentation de Juan Cruz-Benito[9].

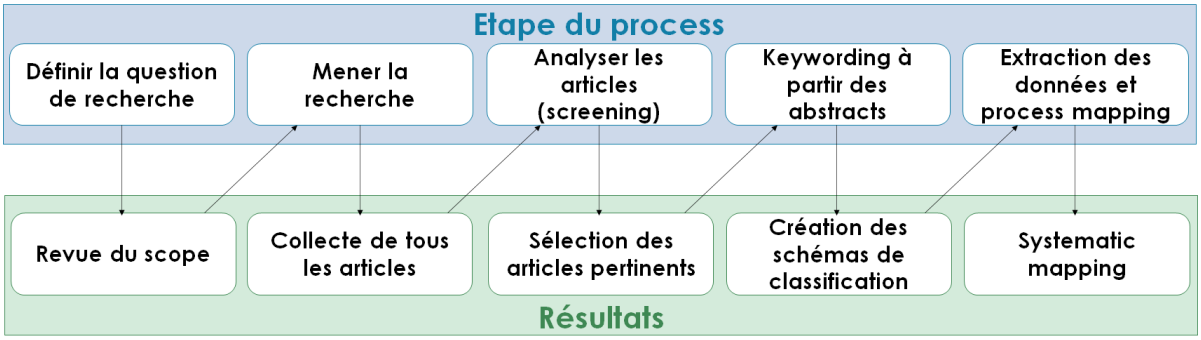


Figure 1: Le processus de mapping study[31]

Plusieurs étapes, présentées dans la Fig. 1, doivent être suivies pour arriver à une carte systématique :

1. **Définition des questions de recherche :** Cette étape est la base qui doit donner la direction de la recherche. Le choix des questions doit être pensé correctement pour arriver à produire une mapping study efficiente.

2. **Conduire la recherche :** cette étape permet de récupérer l'ensemble des articles en lien avec les questions de recherche. Pour faire cela, il est proposé de créer des phrases de recherche en lien avec les questions. Pour arriver à faire ces phrases de recherche, on peut récupérer les mots clefs importants des questions de recherche. Ce qui est une bonne manière de cibler plus précisément les articles pertinents. Ces phrases seront construites autour de ces mots clefs en utilisant des termes de liaison tels que 'AND' et 'OR'. Ces termes permettent d'affiner la recherche.

3. **Filtrage des articles :** Le filtrage d'articles se fait grâce à des critères d'inclusion et d'exclusion. Ceux-ci permettent d'exclure rapidement des articles ne faisant manifestement pas partie du domaine de recherche ou ayant des caractéristiques indésirables. Comme critères, nous pouvons par exemple utiliser la date ou bien le type de document proposé (livre, articles, etc...).

4. **Créations des mots clefs à partir des résumés :** Pour la création des mots clefs, Petersen & al. propose de les créer à partir des résumés lus dans les différents articles sélectionnés précédemment. La lecture du résumé doit permettre de récupérer les concepts reflétant la contribution du papier. Ainsi, après avoir lu tous les résumés, le chercheur aura un ensemble de mots clefs en lien avec le domaine de recherche. Il pourra finalement sélectionner les mots clefs qui l'intéressent et donc les articles liés. La Figure 2 est un représentation schématique de la proposition de Petersen & al.

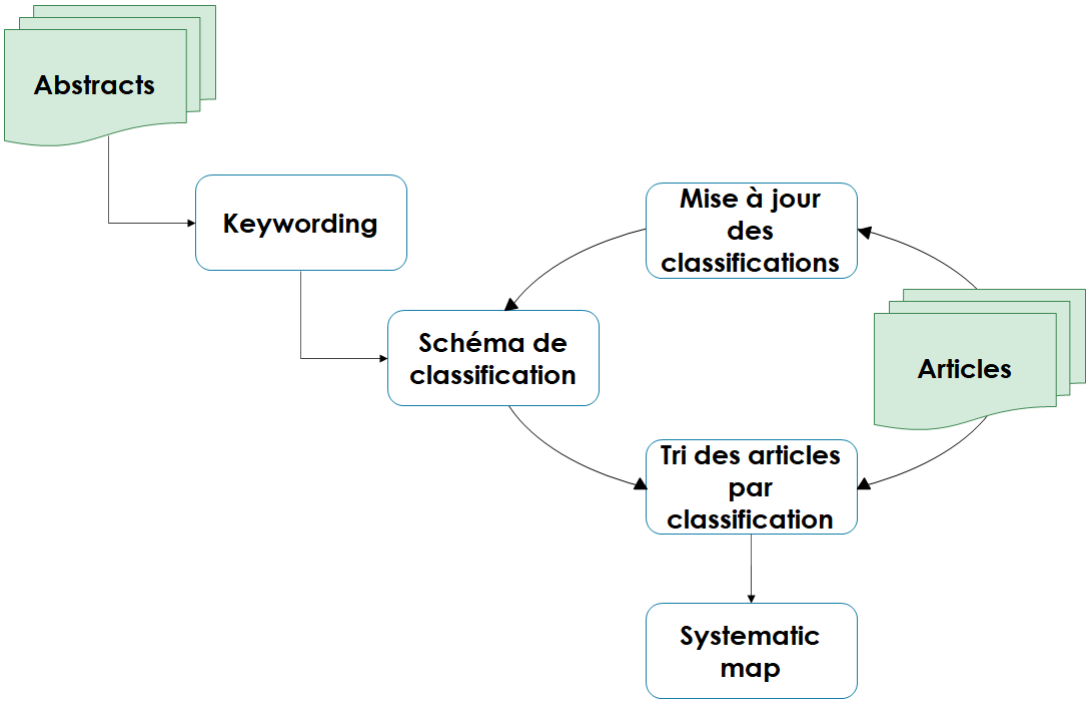


Figure 2: Création des mots clefs[31]

5. **Extraction des données et création du schéma :** Pour cette dernière phase, il sera possible de créer un schéma lié aux articles sélectionnés grâce aux mots clefs. Cette phase pourra répondre aux questions créées précédemment.

L'utilisation d'un mapping study peut être très différente suivant le besoin :

- Rassembler les concepts et les questions clefs en notant les mots, les phrases et les thèmes liés à l'étude.
- Créer des cartes conceptuelles en résumant les résultats importants des revues, livres et papiers.
- Présenter un résumé des revues, livres et papiers trouvés dans une Systematic Literature Review (SLR).

Dans ce mémoire, l'utilisation d'une mapping study a été voulue comme un outil de brainstorming ainsi que pour délimiter la portée de l'étude. En nous posant des questions orientées vers les articles, nous avons pu les sélectionner et en réduire le nombre. Elle nous a aussi servi de porte d'entrée pour ensuite aller vers une SLR avec des questions plus ouvertes sur le domaine. Leurs architectures d'évolution se ressemblant, c'est une pratique courante dans la recherche.

2.3 Test logiciel

Les tests, dans le développement logiciel, sont définis comme un processus d'évaluation ayant pour but de déterminer si le système répond aux exigences spécifiées. Il s'agit d'un processus utilisant la validation et la vérification pour certifier que le système réponde aux exigences fournies par le client.[20, 5]

La validation doit permettre de vérifier que le logiciel fait bien ce que l'utilisateur demande réellement, tandis que la vérification doit permettre d'affirmer que le logiciel est conforme aux spécifications.

Les tests de validation démontrent aux développeurs et aux clients du système que le logiciel répond à leurs exigences, un test réussi démontre donc que le système fonctionne comme prévu.

Les tests de vérification permettent de découvrir les défauts ou les défaillances du logiciel lorsque son comportement est incorrect ou non conforme à sa spécification. Les tests de vérification doivent être menés de manière itérative tout au long du développement du logiciel, afin de s'assurer que celui-ci rencontre bien les spécifications désirées.

L'utilisation des tests se fait en utilisant des données artificielles. Il faut vérifier les résultats de l'exécution pour détecter les erreurs, anomalies et informations des attributs non fonctionnels du programme. Il est très important pour un développeur de toujours se rappeler qu'un test révèle la présence d'erreurs, mais pas leurs absences.

2.3.1 Test de régression

Un test de régression vérifie si le logiciel fonctionne toujours correctement après la correction d'erreur ou la mise à jour du système. Il peut arriver que la correction d'une erreur en entraîne de nouvelles, ce qui est appelé une régression dans le logiciel. La correction d'une erreur, l'ajout d'une nouvelle fonctionnalité, la modification d'une fonctionnalité existante ou la modification d'un composant peuvent apporter une régression. Il est donc intéressant de pouvoir le vérifier grâce à des tests. Les tests de régression sont une activité coûteuse et nécessitent beaucoup de temps quand ils sont effectués manuellement.[25, 40, 51]

Une différence importante existe entre les tests de régression et les tests de développement, les tests de régression représentent une suite de tests pouvant être réutilisés à loisir. Ce qui n'est pas le cas des tests de développement. Pour les développeurs, il existe deux stratégies pour la mise en place de ces tests. Soit l'approche "retest-all" qui consomme du temps et des ressources importantes à chaque génération étant donné qu'on ne trie pas les tests à générer. Soit l'approche Regression Test Selection (RTS), ce sont des techniques de sélection des tests de régression. Celles-ci tentent de réduire le temps nécessaire pour tester un programme en ne sélectionnant que les tests ayant un lien avec le code modifié.

Pour gagner du temps, les développeurs automatisent ces tests qui sont, généralement, exécutés un grand nombre de fois.

2.3.2 Automatisation des tests

L'automatisation des tests [13] est une technique permettant de réduire l'effort nécessaire tout en améliorant l'utilisation des tests grâce à l'emploi d'un logiciel permettant d'automatiser le flux des différents tests. Il permet en plus de comparer les résultats obtenus avec les résultats attendus. Cette technique apporte un gain de temps important car, elle diminue les manipulations qui peuvent être assez laborieuses. Dans de nombreux projets de développement, cette automatisation se fait via une intégration continue.

2.3.3 L'intégration continue

La Continuous integration (CI), autrement dit l'intégration continue, est une technique permettant d'automatiser la compilation, la construction et les tests des logiciels faits par différents contributeurs dans un seul et même projet de développement. Le schéma de la figure 3 représente cette technique, la boucle insiste bien sur le caractère répétitif de cette méthode qui permet de vérifier chaque code ajouté avec des tests. Cette pratique permet aux développeurs de logiciels d'envoyer fréquemment et facilement des changements de code dans un dépôt central. Chaque code ajouté est vérifié automatiquement par le système afin de détecter les erreurs d'intégration. L'intégration

continue permet un gain de temps et une meilleure cohérence du code.[14]

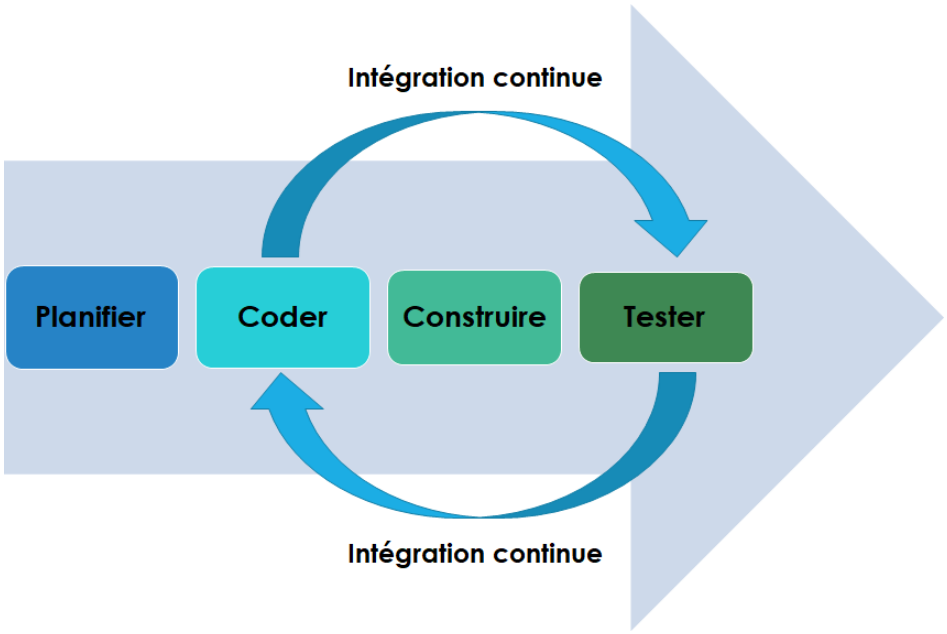


Figure 3: Schéma d'intégration continue[8]

L'utilisation d'un système de contrôle de version du code source, tel que Git est indispensable et doit être complétée par d'autres contrôles, tels que des tests de qualité du code automatisé, des outils de revue du style syntaxique etc.

Sans une CI, la communication entre les développeurs, les testeurs et les architectes peut être compliquée. Ils devront se tenir informés des diverses contributions faites par les autres membres de l'équipe. Il y a un risque de doublons dans le travail et d'erreurs de synchronisation.

La CI permettra de minimiser les tâches d'administration et de communication lors de l'intégration de nouveaux codes. Elle permet aussi d'apporter une aide au développement de flux de travail de qualité. Si un test échoue lors d'une intégration, les développeurs doivent alors identifier la source de l'erreur.

2.4 Débogues

"Le processus de débogage consiste à analyser et éventuellement étendre le programme qui ne répond pas aux spécifications afin de trouver un nouveau programme proche de l'original et satisfaisant les spécifications. Il s'agit donc d'un processus consistant à diagnostiquer la nature précise d'une erreur puis de la corriger." Hailpern & al.[19]

Le point de départ d'une activité de débogage est d'essayer de reproduire une erreur obtenue par un utilisateur dans l'environnement de développement[11]. Quand une erreur apparaît, une exception se produit et un développeur doit essayer de la reproduire

pour comprendre la cause et arriver à la réparer. Une fois effectué, il créera un test qu'il ajoutera dans la liste des tests de régression afin d'être certain que cette erreur ne se reproduise plus lors de futures modifications du logiciel.

Pour la reproduire, il faut arriver à avoir un environnement de débogage suffisamment ressemblant. L'erreur peut être liée au matériel, aux données, à la langue utilisée ... Ce qui rend le travail de reproduction complexe. En plus de cela, il faut que le développeur ait la connaissance suffisante de l'application et des différents composants qui la structurent pour arriver à extraire toutes les informations pouvant amener à cette erreur bien précise.

2.5 Flaky test

Luo & al.[25] définissent le concept de flaky test comme ceci : *"Ce sont des tests dont les résultats sont non déterministes par rapport à une version donnée du logiciel."*

Ces tests créent donc plusieurs problèmes lors des tests de régression :

1. Les erreurs peuvent être difficiles à reproduire en raison de leur indéterminisme.
2. Les flaky tests font perdre du temps aux développeurs. Ils peuvent passer beaucoup de temps à chercher dans le code modifié, alors que ce n'est pas dû au code lui-même.
3. Les flaky tests peuvent cacher de réelles erreurs. Les développeurs arrivent à perdre confiance envers le système et finissent par ignorer certains tests.

Comme présenté dans la section "Test logiciel", les développeurs se basent principalement sur des tests de régression pour vérifier que leurs ajouts n'apportent pas d'erreurs dans le logiciel. Ces tests de régression sont déterministes. Ils sont censés apporter la même réponse à une suite d'exécutions. En apportant cette réponse, les développeurs savent si la fonctionnalité développée apporte une régression. Hélas, les flaky tests sont indéterministes, ils vont donc à l'encontre de la philosophie mise en pratique. De ce fait, vous ne pourrez jamais être certain que vos tests et votre code sont exempts de bogues.

Les flaky tests peuvent apparaître à partir de divers facteurs tels que les incohérences dans les environnements, la mise à jour des données entre les tests, les problèmes de réseau, de temps, etc...

Dans ce mémoire, diverses méthodes proposées par des chercheurs afin de catégoriser et mitiger ces flaky tests seront présentées. Ainsi que des propositions de liens entre ces tests et des méthodes de correction.

3 Développement de la recherche

À la suite de l'état de l'art sur les diverses technologies et méthodes utilisées lors de ce travail, nous allons développer la phase de recherche en tant que telle. C'est ici qu'on va pouvoir présenter les diverses informations récoltées et agrégées pour en extraire des données pouvant répondre aux diverses questions qui ont été posées.

Ce chapitre va nous permettre de vous présenter les différentes questions qui se sont posées lors de la prise en main du domaine et qui ont donné la direction à la méthodologie de travail. Celle-ci s'étant basée sur deux méthodes complémentaires, nous pourrons rapidement répondre aux premières questions qui sont liées à la mapping study. Les réponses des questions liées à la SLR seront quant à elles développées lors d'un chapitre futur après présentation des diverses étapes nous permettant d'y arriver.

3.1 Question de recherche

Durant sa carrière, un développeur sera amené à corriger de nombreux bugs empêchant son logiciel d'être à cent pour cent fonctionnel. Pour cela, il y a déjà de nombreuses solutions qui ont été pensées et développées pour lui rendre la tâche plus aisée. Au-delà de la méthodologie, il y a de nombreux outils permettant de prévoir, détecter, reproduire, corriger les bugs.

Lors de l'utilisation de ces outils ou méthodes, certains tests peuvent se mettre en erreur alors qu'il n'y a pas eu de changement dans le code. Ces tests sont les flaky tests, ils peuvent survenir à tout moment et pour des raisons différentes. Ils apportent de l'instabilité dans le processus de développement des développeurs. Ce qui en fait un sujet important pour la recherche.

Actuellement, quand un développeur est en contact avec un flaky test, sa première action sera de ré-exécuter le test. Grâce à cela, il pourra vérifier si l'erreur survient à nouveau et pourra tenter de trouver la cause de cette erreur. Cette cause peut être très diversifiée et difficile à identifier. Pourtant, l'identifier est la première étape pour arriver à corriger le flaky test. Ce travail va, grâce à la SLR, tenter de trouver les différentes causes possibles et les regrouper en catégories.

Identifier les causes d'un flaky test n'est que la première étape pour arriver à corriger ceux-ci. De nombreuses méthodes de correction et de mitigation ont été étudiées par les chercheurs. Cependant, peu d'informations lient les causes d'un flaky test à ces méthodes. C'est pourquoi ce travail va tenter, grâce aux connaissances actuelles du monde scientifique répertorié dans ce document, de les relier et de proposer le "best choice" suivant la catégorie de flaky test pour les développeurs.

Questions pour la SLR :

- **Question de recherche :** Quelles techniques de debug sont les plus appropriées pour les différentes catégories de flaky test ?

- **Sous-question :** Quelles sont les catégories de flaky test identifiées ?

Pour arriver à répondre à ces questions, la mapping study devra d'abord répondre à ses propres questions. Elles permettront d'obtenir les informations nécessaires pour la SLR.

Questions pour la mapping study :

- **Question 1 :** Combien d'études ont été publiées dans le temps ?
- **Question 2 :** Quels sont les thèmes les plus travaillés pour les recherches ? Les types d'articles ?
- **Question 3 :** Est-ce que le stade des études empiriques a laissé place aux études proposant des solutions ?
- **Question 4 :** Est-ce que de nouveaux domaines se sont intéressés à la question ?

3.2 Méthode de recherche

Lors de la réalisation de ce travail, plusieurs phases ont été suivies. La première fut la découverte du sujet sur base d'un ensemble d'articles identifiés afin de se familiariser avec le domaine et le vocabulaire associé. Cela a permis de sortir une liste de mots-clefs utilisés pour effectuer les phases suivantes.

La seconde fut le choix du type de travail qui allait être proposé. Plusieurs possibilités étaient ouvertes telles qu'une mise en pratique d'un article, une SLR, une Multivocal Literatur Review (MLR), une mapping study etc...

Après réflexion, il a été décidé de faire une mapping study qui serait étoffée d'une SLR. Une mapping study donne une bonne base pour aller ensuite vers une SLR car, comme expliqué, précédemment, dans l'état de l'art, les méthodes ont un démarrage similaire. Par contre, la mapping study se termine plus tôt qu'une SLR et n'apporte pas la possibilité d'ajouter un travail personnel. La SLR quant à elle permet de se poser des questions plus ouvertes sur le domaine étudié ce qui permet l'apport d'idées personnelles. Dans notre cas, elle nous permettra de nous poser des questions sur le debugging des flaky tests.

Ces choix ont été faits pour obtenir un document se basant sur une méthodologie standard et éprouvée.

La méthodologie de la mapping study a été suivie principalement grâce à l'article de Petersen & al.[31] mais comme le disent les auteurs, "le guide le plus suivi n'est pas suffisant à lui tout seul"[32], plusieurs guides ont donc été consultés pour en combiner de multiples directives.

Le but de l'utilisation de la mapping study a été de structurer la zone de recherche, d'identifier les différentes pratiques utilisées par les scientifiques, les différentes variables

prises en compte telles que la taille des programmes, la période de l'analyse, le langage utilisé, l'environnement...

Le résultat de cette mapping study a permis de cartographier le thème, la question dans son ensemble. Il a permis de montrer l'état de la recherche par rapport aux flaky tests avec des graphiques et des tableaux. Grâce à cela, il est aisé de démontrer les grandes tendances dans les recherches menées et ainsi de pouvoir trouver plus facilement les directions encore inexplorées pour de futures recherches.

La SLR a été mise en pratique en suivant la méthodologie présentée dans le travail de Kitchenham & al.[21]. Elle a été choisie, car elle permet de proposer un résumé exhaustif de la littérature en relation avec une question de recherche.

3.2.1 Critères d'inclusion

La sélection des critères d'inclusion a été faite pour obtenir un résultat cohérent avec la question de recherche et une vue suffisamment large dans le temps sur le domaine.

- Sélection d'articles ayant une date de parution supérieure ou égale à l'année 2012. Le sujet n'étant pas encore très développé avant cette date, une fourchette de dix ans a été jugée comme intéressante pour voir l'évolution à travers le temps de ce domaine de recherche.
- Sélection des mots clefs. C'est un critère important dans une SLR et une mapping study. Car, Ce sont ces mots clefs qui vont mener au recensement des articles. Plus le nombre de mots clefs est important et moins nous aurons d'articles se trouvant dans la sélection. Dans cette étude, le mot clef le plus important est l'ensemble de mots : "Flaky test". Cet ensemble a été la pierre angulaire de la recherche d'articles, c'est lui qui faisait le lien avec la question de recherche. Nous avons ajouté d'autres mots clefs par la suite, mais toujours en lien avec le premier, ce qui a permis d'obtenir des articles plus ciblés suivant les sous-thèmes choisis.
- Sélection des types d'articles sélectionnés. C'est un autre critère d'inclusion important. L'article sélectionné doit être jugé digne de confiance. Il a donc été décidé de limiter la sélection d'articles à des articles ayant été revus par des pairs.

3.2.2 Critères d'exclusion

La sélection des critères d'exclusion a été faite pour permettre de réduire le nombre d'articles de manière cohérente et justifiée.

- La langue est un critère d'exclusion fort. Le choix a été fait de nous limiter à l'anglais et au français. Car, comprendre les articles sans recourir aux traducteurs a paru indispensable pour la bonne compréhension de ceux-ci.

- La pertinence des articles et leurs liens avec le thème est un autre critère d'exclusion. Un court paragraphe dans l'article ou une allusion aux flaky tests ne suffit pas à apporter une valeur ajoutée suffisamment importante pour pouvoir sélectionner l'article. Hélas, des informations pouvant être utiles pourraient avoir été négligées suite à cette sélection de critères. Ce risque a été pris en compte et a été accepté en regard de la quantité d'articles déjà sélectionnés.

- Le type de papier sélectionné a aussi été un critère d'exclusion. Il a été décidé d'exclure les livres, mémoires, thèses. Ce sont des papiers qui ne sont pas des articles publiés et donc, à priori, pas revus par les pairs.

3.2.3 Recherche et extraction de contenus

La phase de recherche a été réalisée en plusieurs étapes. Premièrement, une sélection des bases de données accessibles à tous a été faite. Il était particulièrement important de sélectionner des bases de données accessibles facilement, car il faut que le travail puisse être reproduit et vérifié par toute personne souhaitant expérimenter le travail accompli.

Il a donc été décidé d'utiliser quatre bases de données différentes :

- IEEE Xplore Digital Library c'est une base de données principalement axée sur la technologie en général, elle reprend donc beaucoup d'articles en lien avec le développement logiciel, l'infrastructure, la gestion informatique ...

- ACM Digital Library c'est une base de données axée exclusivement sur le domaine de l'informatique.

- Scopus Preview c'est une base de données interdisciplinaires qui a un large catalogue et donc qui peut permettre aux utilisateurs de ne pas louper des articles importants.

- Google Scholar c'est une base de données bien différente des précédentes, mais c'est aussi la plus ouverte, car elle reprend l'ensemble des pages web. Il faut cependant prendre des précautions avec son utilisation, car il n'y a pas de tri effectué au préalable. Toute personne peut arriver à ajouter un document dans cette base de données.

La phase suivante a été de créer les différentes requêtes qui ont été utilisées pour chaque base de données. Pour cela, nous avons sélectionné les mots clés et la période que nous avions choisie dans les critères d'inclusion et nous les avons ajoutés à différents termes étant utilisés dans le domaine. Il a par exemple été décidé de rechercher tous les articles en lien avec "flaky test" et "root cause" ou bien "flaky test" et "empirical study". Ces différentes requêtes ont permis de séparer les articles suivant des zones spécifiques du domaine.

	IEEE Xplore	ACM	Goggle Scholar	Scopus
flaky test	26	78	350	92
Requete si existante	**("All Metadata":"flaky test")**	*[All: "flaky test"] AND [Publication Date: (01/01/2012 TO 01/07/2022)]*	"flaky test"	TITLE-ABS-KEY ("flaky test") AND PUBYEAR > 2011 AND PUBYEAR > 2011
root cause	9	30	124	16
Requete si existante	**("All Metadata":flaky test) AND ("All Metadata":root cause)**	*[All: "flaky test"] AND [All: "root cause"] AND [Publication Date: (01/01/2012 TO 01/07/2022)]*	"flaky test" and "root cause"	TITLE-ABS-KEY ("flaky test" AND "root cause") AND PUBYEAR > 2011 AND PUBYEAR > 2011
empirical study	21	34	219	19
Requete si existante	**("All Metadata":flaky test) AND ("All Metadata":empirical study)**	*[All: "flaky test"] AND [All: "empirical study"] AND [Publication Date: (01/01/2012 TO 01/07/2022)]*	flaky test and "empirical study"	TITLE-ABS-KEY ("flaky test" AND "empirical study") AND PUBYEAR > 2011 AND PUBYEAR > 2011
non-determinism	4	26	102	13
Requete si existante	**("All Metadata":flaky test) AND ("All Metadata":non-determinism)**	*[All: "flaky test"] AND [All: "non-determinism"] AND [Publication Date: (01/01/2012 TO 01/07/2022)]*	"flaky test" and "non-determinism"	TITLE-ABS-KEY ("flaky test" AND "non-determinism") AND PUBYEAR > 2011 AND PUBYEAR > 2011
test flakiness	22	30	114	29
Requete si existante	**("All Metadata":flaky test) AND ("All Metadata":test-flakiness)**	*[All: "flaky test"] AND [All: "test flakiness"] AND [Publication Date: (01/01/2012 TO 01/07/2022)]*	"flaky test" and "test flakiness"	TITLE-ABS-KEY ("flaky test" AND "test flakiness") AND PUBYEAR > 2011 AND PUBYEAR > 2011

Figure 4: Requêtes effectuées dans les différentes bases de données

Pour chaque base de données reprise dans la Fig.4, nous avons fait les mêmes requêtes et nous avons pu voir des différences dans le nombre d'articles récupérés. Nous pouvons d'ailleurs voir dans la figure que le nombre d'articles trouvés dans google scholar est sensiblement plus important que dans les autres bases de données. Ce qui n'est pas spécialement un gage de qualité mais cela prouve que google scholar est plus ouvert à l'ajout de documents.

3.2.4 Sélection des études et évaluation de leurs qualités

La technique de sélection des différents articles récupérés dans les bases de données s'est basée sur les critères d'exclusion précédemment proposés. La langue, les doublons et les articles rétractés ou inaccessibles ont été directement éliminés.

Une lecture des abstracts, conclusions et si nécessaire des introductions a été faite pour la première sélection d'articles. Beaucoup d'entre eux ont été éliminés, car il n'entrait pas réellement dans le sujet et ont été considérés comme du bruit. Le nombre d'articles parait important, c'est le problème quand nous utilisons Google Scholar, le

filtre et le tri n'ont pas été faits en amont et nous devons le faire manuellement.

Le format de l'article a aussi été pris en compte. Les thèses et mémoires ont été systématiquement éliminés de la sélection. Lors de la lecture des articles, la méthode dite du "snowball" a été utilisée. Elle a permis de sélectionner des articles n'étant pas sortis dans les bases de données, mais qui ont été jugés intéressants et en rapport avec la thématique vue ou l'angle d'approche d'un thème ou d'une question spécifique.

Finalement, sur les 576 articles venant des différentes bases de données, 78 ont été sélectionnés.

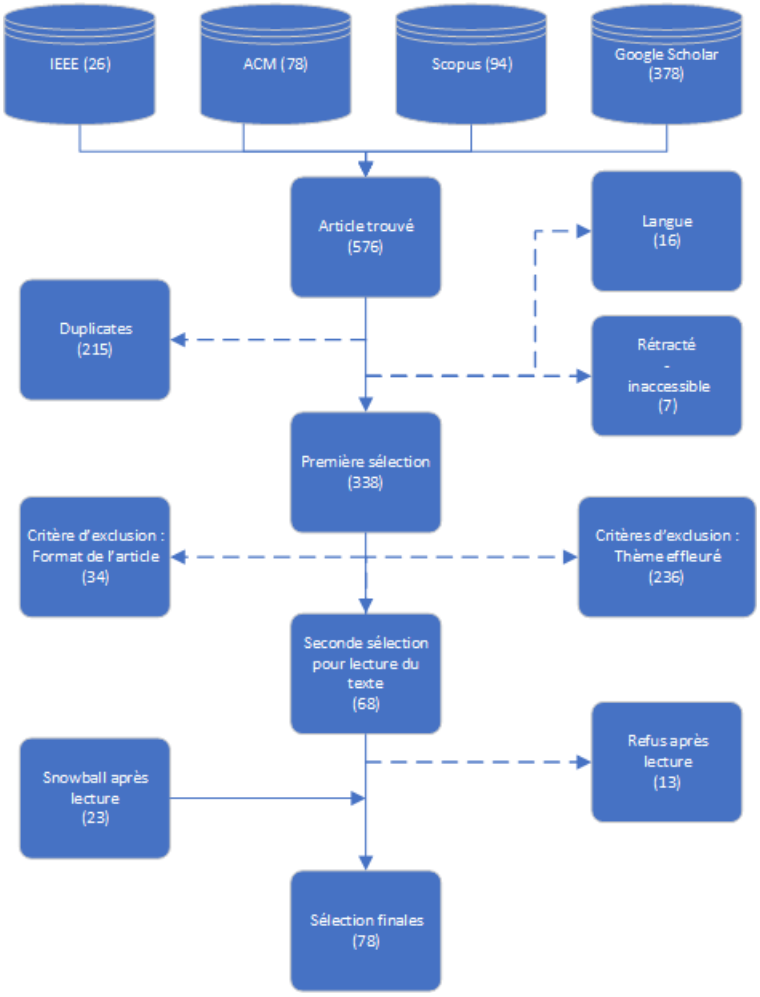


Figure 5: Sélection d'articles

3.2.5 Analyse des résultats

La sélection des articles ayant été faite, le travail d'analyse a pu commencer. Pour la première étape, il a été décidé de récupérer, pour chacun des articles, des informations

spécifiques devant nous servir à analyser de manière factuelle, l'ensemble des données agrégées et ainsi pouvoir répondre aux diverses questions de la mapping study.

Ces diverses informations permettent en plus de pouvoir créer des graphiques permettant de les visualiser autrement. Ce qui s'avère intéressant pour arriver à appréhender des idées complexes ou qui doivent être déduites des chiffres.

Chaque caractéristique a été divisée en sous-catégorie. Ces sous-catégories ont été créées en lisant l'abstract de l'article, la conclusion et, si nécessaire, l'introduction. Les mots clefs ont pu aider aussi à la bonne sélection de la catégorie. Ces choix ont été faits avec un maximum d'impartialité, mais pourraient être différents si l'étude était refaite par une autre personne. Il aurait fallu un plus grand nombre de lecteurs pour avoir une liste plus objective.

Voici le résumé des caractéristiques récupérées :

- **La date :** La date va permettre de connaître les tendances au niveau de la recherche. De savoir si le domaine des flaky tests évolue à travers le temps, car c'est bien celui-là qui a été visé dans la mapping study. (Figure 6)

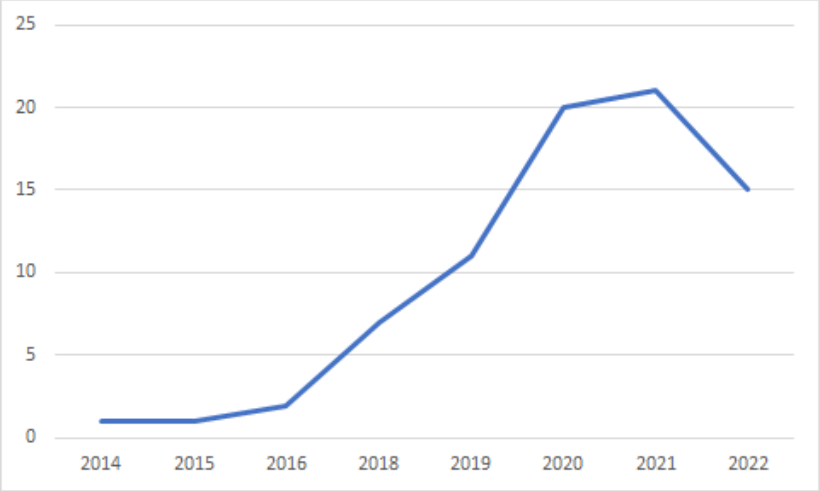


Figure 6: Graphique représentant le nombre d'articles par année.

- **La méthode :** Dans ce cas-ci, le terme "méthode" a été choisi pour parler du type de recherche effectuée par les chercheurs. Toutes les études ne se font pas de la même manière. Il peut donc être intéressant de savoir quels types d'études ont été les plus suivis au fil du temps. Cette évolution permet de déduire des informations sur la maturité du domaine. (Figure 7)

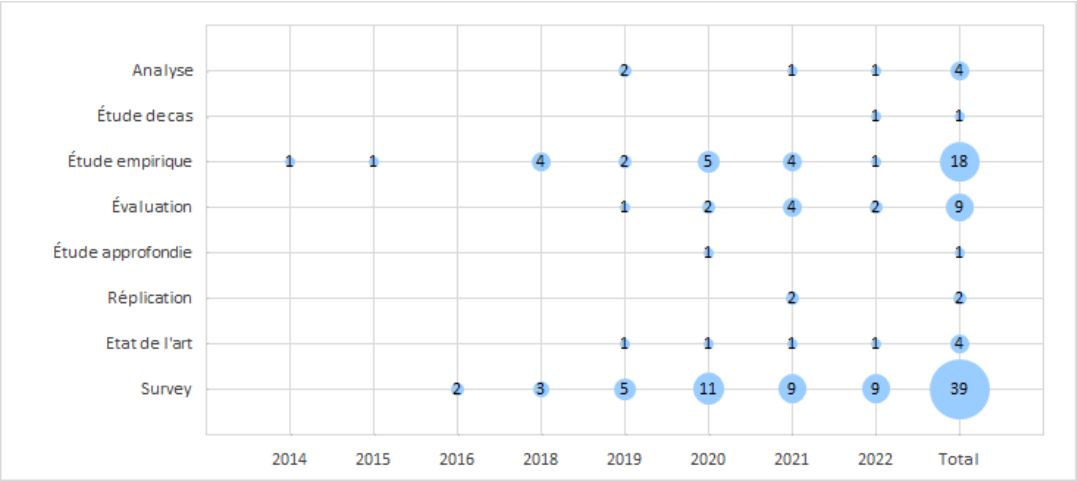


Figure 7: Graphique représentant la méthode utilisée pour écrire l'article.

- **Le type :** Le type, ici, représente plus précisément le domaine visé par l'article. Encore une fois, l'évolution des domaines à travers le temps permet de connaître la direction prise par les chercheurs. Développer des stratégies et outils ne se fait pas au début de la découverte d'un domaine. (Figure 8)

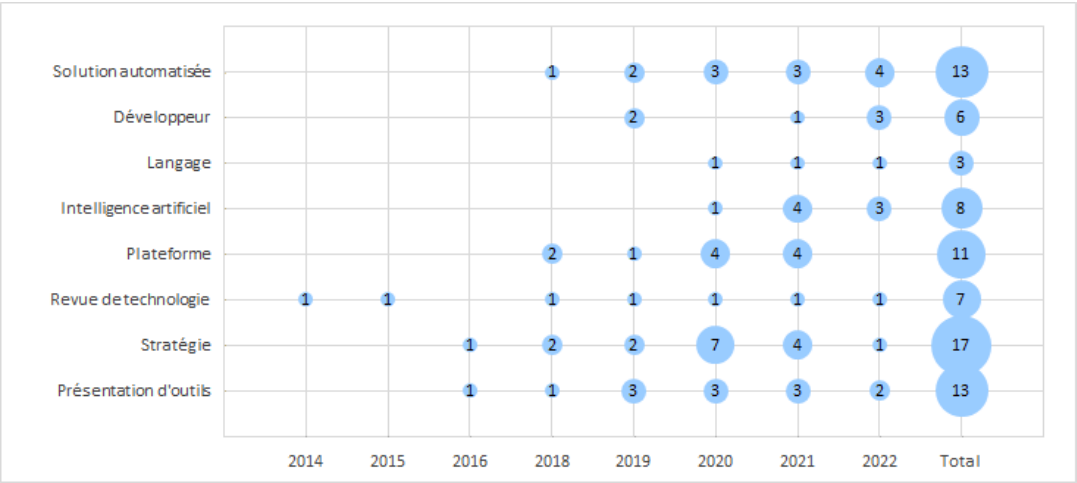


Figure 8: Graphique représentant le type de papier écrit par les chercheurs.

- **Le but ou la technique :** Ceux-ci diffèrent fortement dans chaque article, une tentative de regroupement a été faite pour réduire la taille du graphique et garder une information lisible. Le but et la technique représentent les informations voulant être transmises au lecteur. (Figure 9)

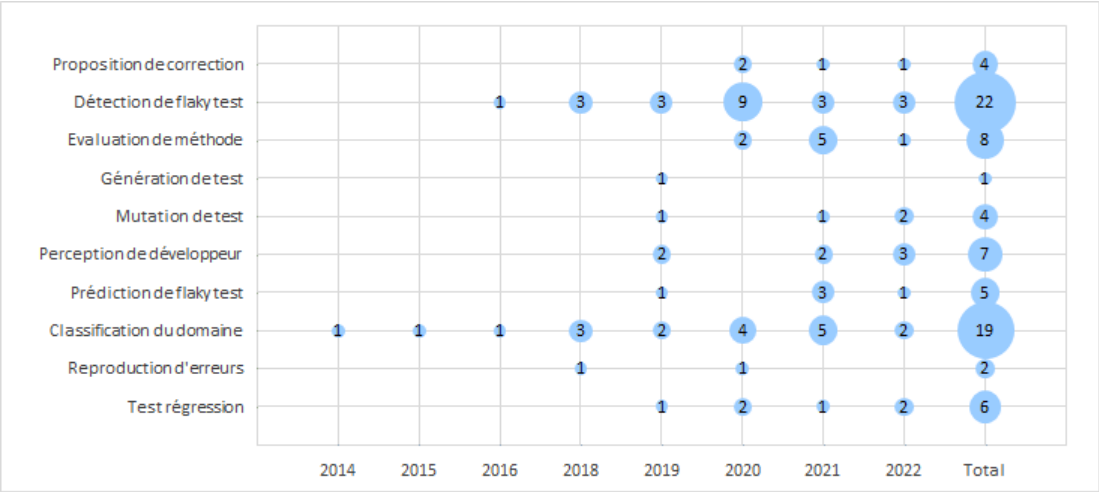


Figure 9: Graphique représentant le but de l'article / la technique utilisée.

Toutes ces informations, sont contenues dans un fichier Excel ayant servi de base de données. Il contient aussi les différents mots clefs utilisés pour chaque article. Ces mots clefs viennent des auteurs si ceux-ci les ont fournis, ou bien de l'auteur de ce travail quand ils n'existaient pas. Les mots clefs ont d'ailleurs aussi servi à la SLR pour sélectionner les articles à lire suivant le domaine visé.

4 Vue d'ensemble

4.1 Proposition basée sur la mapping study

Grâce aux informations récoltées lors de la mise en application de la mapping study, une proposition de réponse peut être développée. Le cadre de ces propositions se limite au cadre mis en place lors de la présentation de la méthodologie de travail. Ceci sous-entend donc que les réponses proposées se limiteront aux informations récoltées. Il se peut que les réponses puissent être discutées si les critères sont modifiés.

Comme dit précédemment, la méthodologie doit pouvoir garantir la reproduction de cette recherche par d'autres personnes et l'obtention de résultats similaires.

4.1.1 Combien d'études ont été publiées dans le temps ?

Pour répondre à la question simplement, il y a eu 78 articles publiés. Nous pouvons cependant obtenir plus d'informations des données recueillies. Pour cela, il faut se baser sur les éléments importants de la question.

Le premier élément important est le temps. Celui-ci a été délimité entre le 01 janvier 2012 et le 01 juillet 2022 date à laquelle les requêtes ont été faites. Cette période de dix ans a semblé être une fourchette adéquate au vu du thème et du nombre d'articles récupérés.

Le second élément important pour cette question est évidemment le thème des études ciblé. Pour cela, la sélection des mots clés a été la caractéristique principale. La recherche s'est basée sur un couple de mots clefs organiques qui est "flaky test" et qui est la base de nos diverses requêtes. Les autres mots clefs ont été utilisés en couple avec le premier ce qui fait que la recherche était affinée mais que la base d'articles restait la même.

En se basant sur la Figure 6, des informations peuvent émerger. Premièrement, nous pouvons faire le postulat d'un démarrage de l'utilisation du terme flaky test à partir de 2014. En effet, notre sélection ne contient pas d'articles écrits avant 2014. Elle ne contient d'ailleurs qu'un seul article cette année là et un seul différent l'année suivante.

Ensuite, nous pouvons observer une augmentation constante du nombre d'articles. Le thème s'est donc étoffé et a pris de l'importance au cours des années suivantes. D'ailleurs, l'année 2022 ne devrait pas faire exception car elle atteint déjà le nombre de 15 articles après seulement 6 mois.

4.1.2 Quels sont les thèmes les plus travaillés pour les recherches ? Les types d'articles ?

Comme le montre la Figure 8, de nombreuses études sur les diverses stratégies pour arriver à gérer les flaky tests ont été proposées. À cela, s'ajoutent de nombreuses propositions d'outils et de méthodes automatisées. On peut voir que l'année 2020 a été

particulièrement propice pour ces domaines et types d'articles

Les diverses plate-formes sont aussi intéressées par le sujet, dans cette catégorie, Android et le web ont le plus d'études en lien avec les flaky tests. Ces plate-formes sont plus souvent utilisées pour les projets open-sources, on pourrait y déceler un certain lien.

Si l'on regarde la Figure 9, on peut voir que la détection de flaky test fait partie des techniques fort étudiées depuis 2018 et surtout en 2020. Divers articles ont été catégorisés dans "Présentation du domaine", ce sont les articles parlant spécifiquement des flaky tests en ce compris leurs identifications, catégorisations, gestions, etc...

4.1.3 Est-ce que le stade des études empiriques a laissé place aux études proposant des solutions ?

Il est clairement perceptible que les études empiriques ont joué un rôle important dès le début des recherches sur le domaine. Elles ont été des précurseurs et sont toujours utilisées lors de recherches faisant des liens vers d'autres domaines ou plate-formes.

Cependant, très rapidement, le nombre de survey a évolué pour représenter la moitié des articles ayant été récoltés. Ces articles sont basés sur les différentes techniques pour arriver à classifier, détecter et corriger les flaky tests. Ils contiennent aussi les enquêtes faites auprès des développeurs pour connaître l'impact des flaky tests dans leur vie de tous les jours.

4.1.4 Est-ce que de nouveaux domaines se sont intéressés à la question ?

Les graphiques présentés ici répondent difficilement à cette question. Mais nous pouvons quand même voir de nouvelles tendances arriver avec par exemple, des études sur la vision des développeurs, de nouveaux langages utilisés et surtout l'utilisation de l'intelligence artificielle. Le machine learning propose déjà de nouveaux algorithmes se voulant adaptés pour la détection de flaky tests et leurs corrections. La mutation de tests existants propose aussi de nouvelles études pour arriver à réduire le nombre de flaky tests.

4.2 Discussion mapping study

D'après ce que nous pouvons voir des différents graphiques présentés, le domaine des flaky tests a grandement évolué durant ces dix dernières années. Que ce soit les méthodes de classification, de détection, de correction ou de reproduction d'erreurs, de nombreuses études ont déjà été réalisées avec de belles avancées. On peut ainsi voir que le nombre d'études réalisées chaque année va en augmentant et se diversifient dans les sujets. Les premières études ont permis à de nombreux chercheurs de proposer des solutions lors de problèmes rencontrés.

4.3 Proposition basée sur la SLR

Le résultat de la mapping study reste très lié aux articles dans leur ensemble et aux informations extractibles de ces graphiques. La SLR va permettre de pouvoir approfondir des thèmes liés aux flaky tests.

Les prochains chapitres vont permettre de faire le tour de la question. De bien comprendre les enjeux des développeurs quand ils sont face à un flaky test. D'avoir un aperçu des différentes méthodes de détection pouvant être mises en place pour arriver à trouver le flaky test. De pouvoir les classifier pour connaître leurs causes et proposer une méthode de résolution.

Un autre enjeu est lié à leur reproduction. La reproduction d'erreurs est un coût indéniable pour le développeur, avoir une idée des solutions à venir permettra à ceux-ci de pouvoir mieux évoluer et d'avoir une meilleur gestion de l'intégration continue.

La SLR se penchera aussi sur les différents challenges, avec des domaines parallèles mais de façon plus succincte. Les liens avec les flaky tests et ces domaines étant actuellement faiblement travaillés.

5 Flaky test et les développeurs

Comme dit précédemment, les flaky tests ne sont pas, encore, très connus en tant que tels par les développeurs. La figure 8 nous montre d'ailleurs que les études qui font le lien entre flaky test et développeurs sont assez récentes (à partir de 2019). Ces études ont visé à travailler de grands axes qui sont :

- La perception d'un flaky test par les développeurs.
- L'impact des flaky tests sur la vie quotidienne des développeurs.
- La méthode de gestion des flaky tests par les développeurs.
- Les besoins des développeurs dans le futur.

Dans ce chapitre, nous allons essayer de faire l'état de l'art de ces grands axes. Pour cela, nous nous baserons sur les différents articles récupérés grâce à notre mapping study.

5.1 Perception d'un flaky test

L'étude de Parry & al.[30] a proposé de définir un flaky test comme suit : "Un flaky test est un test qui peut en même temps réussir et rater sans aucun changement du code pendant la génération des tests." et à cette affirmation, ils ont eu une adhésion de plus de 90% des personnes sondées. Ceci démontre que la base de compréhension commune est bien solide et est communément acceptée.

Par contre, Ahmad & al.[1] ont, dans leur étude, démontré que la perception d'un flaky test est un peu différente suivant le développeur, son historique et le domaine lié à son entreprise. Ils ont proposé de catégoriser les facteurs qui affectent la perception des flaky tests par les développeurs en 4 grandes catégories. Ceux-ci, ont été validés par 86% des personnes sondées.

- **Software test quality :** Cette catégorie contient tous les facteurs ayant un lien avec les tests de qualité. La documentation non tenue à jour, le manque d'informations sur les fonctionnalités faites ou à faire, le manque de connaissance de l'environnement, l'expérience de l'équipe, ... ce sont tous des facteurs ayant un lien avec la perception d'un flaky test. Certains pourront augmenter la difficulté de compréhension d'où le risque d'apparition, alors que d'autres pourront les diminuer.
- **Software Quality :** Cette catégorie reprend les facteurs en lien avec la qualité logiciel. Les fonctionnalités qui modifient en arrière-plan le système, l'infrastructure réseau et matériel, la motivation de l'équipe à vouloir corriger les flaky tests sont autant de facteurs qui ont une incidence sur les flaky tests.

- **Know Flaky test :** Les facteurs liés au flaky test en tant que tel sont principalement liés aux ”code smell”, c'est à dire les mauvaises pratiques de programmation. Mais aussi aux dépendances entre les tests, les ré-exécutions de tests et un meilleur rapport des logs ont une incidence sur la génération de flaky test.
- **Company specific factors :** Pour finir la dernière catégorie est liée aux entreprises elles- mêmes. Chaque société a sa façon de travailler et ses propres facteurs ne sont pas automatiquement applicables. Par exemple, toutes les sociétés n'ont pas inscrit en dur, des informations dans leurs codes ou bien effectué des tests pour vérifier qu'ils ne sont pas flaky.

5.1.1 Conclusion sur les facteurs

Au final, il est important pour une société, une équipe et même un développeur seul, d'avoir une bonne connaissance des différents facteurs. Ils permettent d'améliorer le code créé, de diminuer les risques de faire apparaître des flaky tests, mais aussi d'arriver à les identifier plus rapidement et facilement. Tous ces points pris en compte permettent un gain de ressources matérielles, humaines et financières non négligeable.

5.2 Impact sur la vie quotidienne du développeur

D'après Eck & al.[12], les flaky tests ont un impact non négligeable sur la vie d'un développeur. Ils sont 79% à trouver qu'un flaky test pose un problème modéré à sévère quand ils en rencontrent un et 58% des développeurs sondés en rencontrent au moins une fois par mois.

Dans les divers articles consultés, les impacts sur les développeurs sont nombreux. Certains impacts peuvent paraître évidents :

- **Perte de temps :** Quand un test apparaît comme étant flaky, le développeur va devoir vérifier et passer plusieurs étapes pour arriver à identifier ce qui le rend flaky.
- **Perte de ressources :** Refaire ces tests demande énormément de ressources qui ne peuvent être utilisées pour de nouveaux tests. Cela impacte tous les développeurs qui, au mieux, doivent patienter plus longtemps pour avoir leurs résultats, au pire complètement éviter la phase de test pour être dans le timing du projet.
- **Perte financière :** La perte de temps, de ressources provoquent indubitablement une perte financière. Le salaire des développeurs, le coût de l'énergie, l'utilisation du matériel et du réseau sont autant d'éléments devant entrer en compte dans le calcul financier lié au développement et à la maintenance du code.

- **Augmentation des rapports d'erreurs :** dans leur étude, Rahman et Rigby [36], montrent que les versions de Firefox contenant des flaky tests reçoivent plus de rapports de crash que les versions normales.

Mais d'autres sont plus surprenants :

- **Perte de confiance dans les tests :** Les développeurs qui sont confrontés à de nombreux flaky tests finissent par perdre confiance dans les tests en eux-mêmes et peuvent arriver à ignorer certains tests défaillants au lieux de les corriger. Cela peut entraîner des publications de codes avec des flaky tests et même pire, avec des erreurs qui ne sont pas dues à un flaky test.
- **Ignorer les flaky tests :** Accepter des flaky tests si le système est reconnu comme non fiable.
- **Le développeur peut ne pas être certain d'avoir corrigé le test :** Un flaky test est, de par sa nature, instable. Si le développeur n'a pas bien identifié le problème, il peut penser l'avoir corrigé s'il n'y est plus confronté durant les tests suivants.

5.3 Gestion des flaky tests par les développeurs

Face à ces tests, les développeurs ont actuellement diverses stratégies possibles à leurs dispositions. Habchi & al.[17] ont listé dans différentes catégories, certaines de ces stratégies.

Des stratégies de prévention :

- La mise en place d'une infrastructure de test fiable.
- Définir des procédures de tests, une méthodologie cohérente et qui évite les "code smell" par exemple.
- Limiter les dépendances externes.

Des stratégies de détection :

- Faire fonctionner le test à nouveau et aussi dans de nouveaux environnements.
- Faire une analyse manuelle
- Vérifier l'historique des résultats précédents
- Vérifier si le test fait partie des modifications

Des stratégies de correction :

- Corriger sa cause d'instabilité
- L'ignorer
- Le mettre en quarantaine
- Le supprimer des tests
- Le documenter

Des stratégies de maintenance :

- Avoir un système de monitoring et d'historique
- Établir un flux de tests qui protège l'intégration continue

5.4 Utilisation d'outils

Pour s'aider dans la correction de flaky tests, les développeurs cherchent à mettre en place de nouveaux outils. Ceux-ci ont beaucoup d'intérêt, tel qu'automatiser la détection, la classification, la correction ...

Il faut cependant faire attention car un outil peut apporter d'autres problèmes qui pourraient ne pas être anticipés.

Par exemple :

- Le flaky test ne pourrait pas être détecté lors de l'utilisation de certains outils. L'utilisation de méthodes et procédures augmente le temps de génération des flaky tests. Cette augmentation de temps peut faire en sorte que le test ne se mette plus en erreur. Ainsi, il n'est plus possible de reproduire l'erreur et donc de la corriger.
- L'utilisation d'outils, surtout si leur nombre est grandissant, ajoute une demande de ressources non négligeable. Il est donc important de faire une sélection réfléchie de ces outils lors de l'intégration continue.

5.5 Souhaits pour le futur

Dans les différentes recherches menées durant ces quatre dernières années, les chercheurs ont demandé aux développeurs sondés quels étaient leurs souhaits, leurs besoins pour le futur. Cette question est importante car elle permet d'avoir des idées sur leurs besoins, sur de futurs axes de recherches.

- **Meilleure connaissance des flaky tests :** Pour savoir comment corriger un flaky test, un travail d'analyse est important. Connaître ses délimitations, ses

impacts et ses liens avec les autres tests vont permettre de pouvoir le corriger de la façon la plus efficace possible. Il est donc important que les recherches continuent dans ce sens. Une meilleure connaissance des ”root cause” permettra de proposer des outils ayant comme principal objectif de surveiller les facteurs connus liés à ces ”root cause”.

- **Une meilleure visualisation des flaky tests :** Un monitoring des tests avec un historique fourni permet de pouvoir comparer les informations et les résultats sans devoir refaire le test. Cette demande produit un gain de ressources et de temps non négligeable pour les développeurs.
- **Des pluggins IDE de détection automatique :** pour leur travail de tous les jours, des pluggins intégrés dans leurs IDE leur permettraient d'avoir des alertes automatiques lors de la génération de codes. Pour cela, il est important que les différentes catégories de flaky tests soient bien délimitées et connues. Ensuite, la proposition d'outils pouvant répondre à ces demandes spécifiques peut être imaginée.
- **Des outils de reproduction d'erreurs :** si un test semble être flaky, il est important de pouvoir recréer l'environnement présent durant l'erreur. C'est pourquoi grâce à un historique bien documenté, il serait faisable de créer des outils permettant de générer, dans les mêmes conditions, le logiciel et ainsi reproduire l'erreur.
- **Des outils de correction automatique :** Ce serait un gain de temps important d'avoir des outils qui feraient de la détection de la root cause, de la localisation de l'erreur et de la découverte de la cause de failure.
- **Un guide et de meilleures pratiques :** C'est une demande assez légitime. Il faut créer une documentation ayant pour but de regrouper les informations connues, les outils permettant de prédire, détecter, corriger les flaky tests. Cette documentation aurait une valeur importante pour tous les développeurs. Elle permettrait de donner une ligne de conduite pour identifier la catégorie de flaky tests et ainsi pouvoir répondre de la manière la plus adéquate possible.

6 Méthode de détection

La gestion d'un flaky test peut se faire de différentes manières. Le développeur peut décider d'agir pro-activement contre les flaky tests. Il peut dans ce cas essayer de détecter les parties de codes pouvant amener de l'instabilité dans le processus. Pour faire cela, il existe dans la littérature des études et recherches proposant des outils et des méthodes ayant pour but de prédire ou détecter les flaky tests.

Dans ce chapitre, nous allons présenter certaines de ces études et certains outils ayant l'ambition de proposer des solutions contre les flaky tests. Nous verrons que toutes les études et outils ne se valent pas. La nature des flaky tests oblige une vision large des différentes méthodes de détection de ceux-ci. Les chercheurs sont obligés de se mettre des limites. Au niveau de la technologie employée, de la surcharge engendrée, du langage visé, de l'espace de stockage réservé ...

Ce chapitre a été divisé en sous-chapitres ayant pour objectifs de proposer les grandes voies possibles dans la prédiction et la détection de flaky tests. Ce choix a été fait pour arriver à présenter de façon structurée, les différentes méthodes proposées. Il ne faut pas perdre de vue qu'un flaky test n'est pas un élément fixe et il peut survenir de bien des manières différentes.

L'ambition de ce chapitre n'est pas d'être exhaustif sur toutes les méthodes proposées et existantes mais bien d'ouvrir la compréhension du lecteur à la difficulté qu'amène l'envie de prédire et de détecter des flaky tests.

6.1 Prédiction de flaky tests

La gestion de flaky tests représente un coût conséquent, de la détection à la correction, il y a de nombreuses étapes pouvant chacune être coûteuse en temps et en énergie, autant pour le développeur que pour le système. C'est pourquoi certains chercheurs ont eu l'idée d'essayer de prédire les flaky tests et ainsi d'agir préventivement sur ceux-ci. En agissant de la sorte, ils partent du postulat que le coût de correction sera moindre car le développeur connaîtra mieux le code écrit récemment et pourra le corriger plus rapidement.

Pour arriver à prédire ces futurs flaky tests, ils ont proposé différentes méthodes qui vont de l'utilisation d'intelligence artificielle à la création automatique de tests permettant de révéler les fragilités de ceux écrits par les développeurs.

Nous allons analyser certaines de ces propositions et ainsi présenter leurs forces et faiblesses.

6.1.1 Présentation de Static Test Flakiness Prediction

L'utilisation de machine learning a été proposée pour prédire et classifier les flaky tests, plusieurs chercheurs l'ont d'ailleurs déjà utilisé [33, 18] et de nombreuses méth-

odes de classification existent telles que Decision Trees[16], Naive Bayes[41], Multilayer Perceptron [28], Support Vector Machine[26] ...

Inspirés par ces différents travaux, Pontillo & al.[34, 35] ont proposé une méthode en utilisant seulement des métriques calculées de manière statique.

Pour faire cela, une liste de 25 métriques statiques a été proposée. Les métriques sont des différences entre les tests flaky et non-flaky. La liste a été utilisée avec une méthode de machine learning et le résultat a démontré que ces métriques pouvaient être utilisées pour caractériser un flaky test.

Grâce à ces résultats, il est donc possible d'identifier des flaky tests en ne considérant que le design des tests. Ce qui permet une diminution de la complexité par rapport aux différents modèles de classification proposés précédemment.

6.1.2 Présentation de FlakeFlagger

Un flaky test peut être très difficile à reproduire. D'ailleurs, Alshammari & al.[2] ont expliqué avoir ré-exécuté les tests provenant de 24 projets plus de 10.000 fois chacun.

Ils ont démontré que même avec ce nombre de ré-exécutions, certains n'avaient pas été ré-identifiés comme des flaky tests. C'est assez problématique car il n'est pas envisageable de les ré-exécuter autant de fois dans la vie courante.

C'est pourquoi, ils ont proposé une nouvelle approche qu'ils ont nommée FlakeFlagger et qui récolte un ensemble d'informations de chaque test. Grâce à celui-ci, il va pouvoir prédire, avec un bon taux de réussite, ceux qui pourraient être flaky.

Pour y arriver, il va regarder la ressemblance entre les fonctionnalités du test vérifié et celles de sa base de données. Si il trouve des tests catégorisés comme flaky et ayant un bon niveau de ressemblance, il pourra étiqueter le test vérifié comme flaky. Ils ont démontré que leur approche était équivalente à celles classifiant des flaky tests avec, en revanche, moins de faux positifs. Ce qui est un avantage tout comme le fait qu'il ne ré-exécute pas les différents tests.

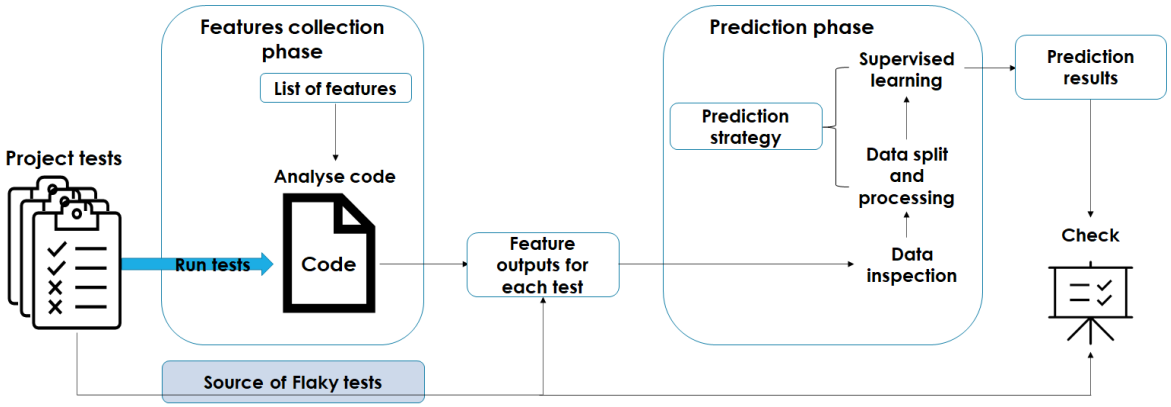


Figure 10: Fonctionnement de FlakeFlagger[2]

La figure 10 représente le fonctionnement de FlakeFlagger dans les grandes lignes.

Pour commencer, une phase de récolte d'informations commence. Pour chaque test, une série de caractéristiques, les appels aux différentes Application Programming Interface (API), une analyse du code et les descriptions sont récoltées et stockées dans une base de données.

Ensuite, grâce aux diverses informations récoltées, FlakeFlagger va vérifier si il trouve des éléments laissant penser que le test pourrait être flaky. Pour cela, une liste de seize caractéristiques pouvant amener à des tests défaillants a été proposée dans l'article sur base d'autres études telle que [25, 1]. Ils ont créé et implémenté des détecteurs pour chacune de ces caractéristiques.

Il utilise aussi un framework hybrid static et dynamic spécifiquement développé pour collecter les instructions utilisées pour chaque test. Il les analyse statiquement afin d'en obtenir les caractéristiques comportementales. Ce framework a été implémenté comme une extension de Maven pour réduire la configuration nécessaire. Il a été fait en Java mais a été pensé pour être généraliste et pourrait être mis en place sur d'autres langages de programmation.

Pour finir, grâce à toutes les informations récoltées, il tente de les classifier avec une fourchette de valeur entre 0 et 1. Plus celle-ci se rapproche de 1 plus la chance que le test soit flaky est élevé.

6.1.3 Présentation de FITTER

Parry & al.[29] ont proposé une technique prénommée Flakiness inducing test creation and repair (FITTER) qui utilise la technique AUTOMATED PROGRAM REPAIR (APR) qui permet de mettre en évidence et d'aider à la correction de sources de flakiness (sous entendu les parties de code pouvant amener à une erreur flaky).

Lors de la présentation de l'étude, les chercheurs s'étaient appliqués à travailler sur deux sources principales d'erreurs qui sont "les dépendances d'ordre et les pertes de ressources". Ils ont donc proposé une technique qui vise à générer automatiquement un Flakiness-inducing test (FIT) qui est un test générant l'erreur dans les tests proposés par les développeurs.

Pour les chercheurs, en proposant un FIT au développeur, celui-ci comprendra plus aisément la cause de son erreur et aura plus d'informations pour la corriger. D'autant plus que ce test aura été écrit très récemment.

Pour réaliser les FIT et vérifier qu'ils font bien ce qui est attendu, FITTER va effectuer trois types d'évaluation sur ceux-ci.

1. Vérifier le nombre d'opérations de modifications effectuées par le FIT sur les objets mutables. Ce type d'évaluation permet de vérifier les erreurs liées à l'ordre des tests.

2. Vérifier les assertions dans les tests. Ce type d'évaluation permet de vérifier si une assertion qui est habituellement vraie peut facilement devenir fausse.

3. Vérifier le chemin suivi habituellement lors du test. Ce type d'évaluation permet de vérifier si un changement de celui-ci est possible et facilement faisable.

Les deux derniers types d'évaluation sont calculés avant et après la mise en place du FIT, ce qui permet de différencier les résultats.

À la différence des autres techniques, FITTER génère automatiquement des tests pour détecter les faiblesses des tests utilisés par les développeurs. Cela permet qu'ils utilisent peu de ressources car il se concentre sur de petites portions de code au lieu de régénérer un ensemble de tests. Un autre avantage, est qu'il peut être utilisé avec différents langages, ce qui le rend très intéressant.

6.2 Ré-exécution des tests

La méthode la plus utilisée pour détecter les flaky tests est la "ré-exécution". Le but est de ré-exécuter chacun des tests en erreur de multiples fois. Si le test passe ne serait ce qu'une seule fois, alors le test est classifié comme un flaky test. C'est évidement le même principe dans l'autre sens. Si il passe plusieurs fois mais qu'une fois il échoue alors il est aussi classifié comme Flaky test.

Une des problématiques de cette méthode vient du fait qu'il faut un nombre inconnu de tentatives avant de savoir si le test est flaky ou pas (Lam & al.[24] ont d'ailleurs, dans leur étude, ré-exécuté leurs tests 4000 fois chacun et un de ceux-ci n'a été en erreur qu'une seule fois lors de toutes ses ré-exécutions. En effet, la seule autre catégorisation possible est la catégorisation "inconnue" car, nous ne pouvons pas certifier qu'il n'échouera pas lors de l'exécution du test suivant.

Une autre problématique de cette méthode, est le coût d'utilisation autant en termes de ressource serveur que de ressource développeur. Faire tous ces tests prend beaucoup de puissance et de temps.

6.3 Outils avec ré-exécution des tests

6.3.1 Flaky test order-dependent ou non-order-dependent

Lam & al.[23] ont proposé un outil de détection des flaky tests qui propose de les classifier si ils sont Order-dependent (OD) (qui dépend de l'ordre dans lequel ils sont exécutés dans la suite des tests) ou Non-order-dependent (NOD).

Il est utile de savoir si un flaky test est OD car les développeurs peuvent mettre en place des suites de tests raccourcies ou définies pour gagner du temps et vérifier une fourchette de fonctionnalités réduites. Le framework de tests peut aussi changer aléatoirement l'ordre des différents tests même quand il fait tourner l'ensemble des tests.

iDFlakies

Cet outil se limite à classifier les flaky tests en OD ou NOD. Il prend comme information d'entrée une suite de tests, la configuration d'ordre de passage des différents tests et le nombre de fois qu'il doit faire tourner les tests. Les informations de sortie seront une liste des flaky tests avec son type (OD ou NOD) et l'ordre exact utilisé pour obtenir les flaky tests.

Pour arriver à classifier les flaky tests, l'outil fonctionne par étapes. Si une erreur apparaît pour la première fois, l'outil refait l'ensemble des tests dans l'ordre par défaut et après, il les refait dans l'ordre ayant généré l'erreur. Si le test rate encore dans l'ordre ayant généré l'erreur, mais qu'il fonctionne dans l'ordre original, alors il est étiqueté comme un test OD. Sinon, c'est un NOD. Une classification n'est pas définitive au premier passage. L'outil utilise un système de pourcentage pour vérifier à nouveau certaines erreurs. Si le pourcentage est à 100 alors chaque erreur sera vérifiée à nouveau. Si il est à 0 alors aucune erreur ne sera vérifiée à nouveau. Un test classifié en NOD restera comme cela même si il devient OD dans un autre ordre de test.

L'outil créé par Lam & al. a été fait avec comme objectif de pouvoir comprendre le ratio entre OD et NOD dans les projets Open sources et aussi permettre aux développeurs les compromis potentiels entre les configurations et ainsi mieux utiliser leurs ressources.

6.4 Outils sans ré-exécution des tests

6.4.1 DeFlaker

Au vu du coût important de la ré-exécution des tests, Bell & al.[4] ont proposé un outil, "DeFlaker", permettant de savoir si le test est flaky sans avoir à le ré-exécuter. Pour y arriver, il fonctionne en trois phases bien distinctes.

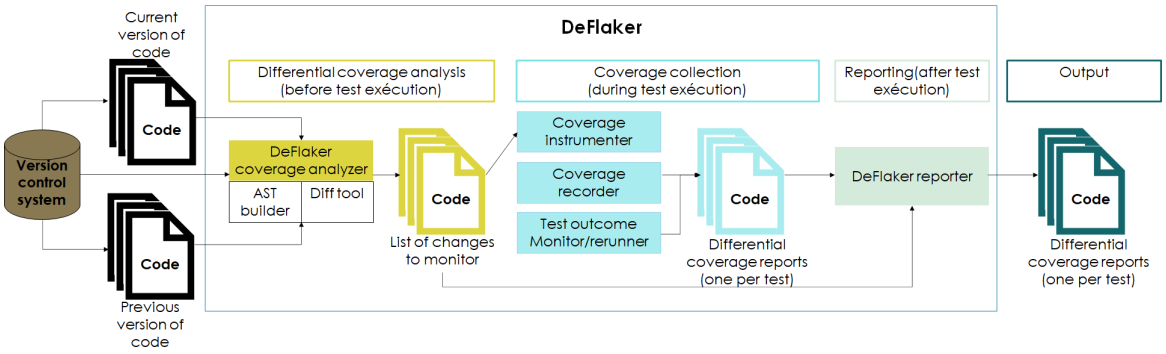


Figure 11: Fonctionnement de Deflaker[4]

Premièrement, il analyse les différences entre les couvertures du code. Il utilise un ABSTRACT-SYNTAX TREE (AST) et un Version Control System (VCS) pour identifier une liste de changements à suivre dans chaque fichier source. Grâce au VCS,

DeFlaker identifie les fichiers modifiés, supprimés ou ajoutés. Il crée ensuite un AST pour chaque fichier créé ou modifié. Pour les nouveaux, il indique chaque type tel que : classe, interface, enum ...

Deuxièmement, il s'ajoute dans le processus de test pour surveiller chaque changement identifié dans les fichiers sources, effectué lors de la phase précédente.

Pour finir, DeFlaker analyse les informations et les résultats pour déterminer quels tests sont possiblement flaky. Il fournit un rapport regroupant toutes les informations nécessaires à la bonne compréhension de chaque test.

Cette méthode permet, grâce à l'historique des changements, de savoir si une partie de code apportant l'erreur dans les tests a été changée lors de la dernière génération du code. Cette information est importante car elle permet de déduire avec une grande probabilité si le test est flaky ou pas. Cet outil doit travailler avec un historique bien documenté sinon il ne pourra pas fonctionner correctement.

Un des points forts de l'outil est sa facilité d'implémentation. Il suffit d'insérer 7 lignes dans le fichier de configuration de Maven pour qu'il soit fonctionnel. C'est non-négligeable quand on connaît le gain de temps possible.

Un autre point fort est la faible demande en ressources nécessaires à son fonctionnement. Au vu des différents tests fait précédemment il n'y aurait qu'une augmentation de 4,5%. Ce qui est dérisoire au vu des gains possibles car les développeurs ne doivent plus ré-exécuter les différents tests en erreur pour savoir si ils sont flaky ou pas.

6.4.2 ADINS & NONDEX

Dans la littérature, les propositions de méthodes de détection de flaky tests sont nombreuses. Cela s'explique par la nature des flaky tests et leurs multiples possibilités d'apparition. Une des propositions vient de Shi & al.[42] qui nous parle de code Assumes a Deterministic Implementation of a Non-deterministic Specification (ADINS). Cela représente un code qui suppose une mise en oeuvre déterministe d'une méthode ayant une spécification non-déterministe.

Dans l'étude, ils présentent un exemple d'*ADINS* avec l'utilisation d'un *HashSet*. La documentation Java[27] définit un *HashSet* comme ceci : " Il ne donne aucune garantie quant à l'ordre d'itération de l'ensemble ; en particulier, il ne garantit pas que l'ordre restera constant dans le temps." Lors de son utilisation, si le développeur compte sur le fait qu'il sortira les éléments dans un ordre précis, il créera un *ADINS* et rendra le code potentiellement flaky.

L'étude propose de détecter ce genre de méthodes en proposant un outil se nommant *NONDEX*. Pour qu'il soit fonctionnel, les chercheurs ont dû suivre plusieurs étapes. Premièrement, ils ont dû identifier les méthodes ayant des spécifications non-déterministes.

Pour faire cela, ils ont utilisé deux requêtes basées sur les mots clefs javadoc et les types de retour. La première requête se basait spécifiquement sur les mots clefs

pouvant indiquer des spécifications non-déterministes telles que : "ordre", "déterministe" et "non spécifié". La seconde requête cherchait des méthodes publiques ayant un tableau comme retour. Au vu des résultats, les chercheurs pensent que la méthode pourrait être améliorée et devra faire l'objet d'un travail spécifique. Ils ont cependant pu obtenir des méthodes non-déterministes qui peuvent servir à *NONDEX*.

Au vu des méthodes trouvées, ils ont pu en extraire trois grandes catégories :

- **Random :** La spécification de la méthode ne donne pas le type de retour. On ne peut donc pas compter sur un retour spécifique de la méthode.
- **Permute :** Le retour de la méthode n'a pas d'ordre spécifique. On ne peut donc pas prévoir l'ordre des éléments.
- **Extend :** La longueur spécifiée est inférieure à la longueur précise.

Après avoir identifié les différentes méthodes pouvant amener un *ADINS*, les chercheurs ont proposé des modèles servant à explorer le potentiel non-déterministe autorisé par la spécification des méthodes. Chaque modèle a quatre différents niveaux de non-déterminisme (*FULL*, *ID*, *EQ*, *ONE*) et une méthode pourrait ou pas les atteindre suivant la version du java utilisé.

Ces niveaux décrivent les cas amenant la méthode à être non-déterministe et permettent à l'outil de diagnostiquer si cela entraînera une erreur flaky ou pas.

NONDEX a été testé avec le langage Java mais il pourrait être utilisé par d'autres langages. Il permet de détecter certains types de flaky tests sans devoir ré-exécuter les différents tests ce qui est un avantage certain et pourrait être utilisé pro-activement lors du développement des logiciels.

6.5 Ce qu'il faut en retenir

Lors de ce chapitre, nous avons pu découvrir différentes méthodes pour arriver à gérer en amont un flaky test. La prédiction et la détection sont des thèmes de recherche vastes et complexes, ce qui en fait des challenges intéressants pour le futur.

Pour ce qui est de la prédiction, différentes propositions ont été présentées :

- L'utilisation de l'intelligence artificielle, par exemple, promet de bons résultats mais nécessite une base de données déjà fournie en exemples de flaky tests ainsi qu'un algorithme efficace. Les différents travaux réalisés sur le vocabulaire des flaky tests[18, 6, 33] sont d'ailleurs une bonne base de travail.
- FlakeFlagger[2] est un outil qui récolte un ensemble d'informations sur chaque test et permet de les comparer pour prédire suivant la ressemblance des tests si il y a des risques d'être flaky.

- FITTER[29] propose de détecter des méthodes pouvant apporter de la flakiness au test. Il crée automatiquement un test ayant pour but de générer l'erreur flaky du test.

Ces différentes propositions apportent des réponses. Hélas, l'impact est réduit à leurs zones d'effet. Actuellement, les root causes visées sont relativement restreintes et il faudrait mélanger un certain nombre de méthodes pour arriver à obtenir un résultat efficace pour les développeurs. La recherche et les propositions doivent encore évoluer pour que la prédiction soit suffisamment aboutie et qu'un développeur puisse y placer un degré de confiance acceptable.

Pour ce qui est de la détection des flaky tests, deux voies ont été suivies. Des propositions ont été faites en ré-exécutant les tests, d'autres ont été faites avec l'objectif de réduire un maximum ces ré-exécutions. La vision derrière ces choix est guidée par le coût que représente la ré-exécution.

Actuellement, la ré-exécution des flaky tests représente un coût conséquent. Dans certains cas, il faut les ré-exécuter un grand nombre de fois pour arriver à reproduire l'erreur. Un autre souci est que les outils sont aussi développés pour arriver à reproduire des root causes spécifiques. Pour les développeurs, il serait intéressant d'augmenter ce nombre de root causes afin de réduire le nombre d'outils et ainsi d'augmenter le niveau d'adhésion de ceux-ci.

7 Root cause

Quand un développeur découvre un flaky test, il va y avoir plusieurs étapes à suivre avant de pouvoir le résoudre. S'il ne suit pas ces étapes, il risque de ne pas corriger de manière pérenne le test. Il risque de déployer une correction qui ne répond qu'à une partie de l'erreur ou qui crée de nouvelles possibilités d'erreur. Il est donc important qu'il suive une méthodologie précise pour arriver à la meilleure correction possible.

Pour commencer, il faut arriver à identifier la cause de l'erreur. "*Qu'est ce qui fait que mon test n'a pas réussi cette fois ci ?*" Identifier cette cause permettra d'en déduire la méthode de correction la plus adaptée. Dans la figure 9 nous pouvons voir que la recherche et la classification des root causes ont, dès le début, été une voie suivie par les chercheurs. De nombreux articles catégorisent les différentes causes. Ce thème est, d'ailleurs, souvent abordé dans les recherches en lien avec les flaky tests.

7.1 Classification des flaky tests

L'article de Luo & al.[25] est considéré comme l'article de référence dans la catégorisation des flaky tests. Cet article présente l'étude de 201 commits ayant corrigé des flaky tests dans 51 projets open-source. Ils ont fait une liste des catégories se retrouvant le plus souvent dans les commits. Celle ci contient 10 catégories différentes.

Bien qu'il soit considéré comme un article de référence, beaucoup d'autres articles ont mis à jour de nouvelles catégories, ce qui permet d'améliorer les connaissances liées aux flaky tests. Il n'est donc pas inutile de les rassembler pour en faire l'état de l'art.

Chaque catégorie de root cause a été étudiée à divers degrés. Certaines catégories ont déjà des propositions de correction, avec des méthodologies et des outils. D'autres n'en sont pas encore à ce stade, car peuvent être plus complexes à corriger.

Pour regrouper les différentes catégories, il a été décidé de suivre la proposition de Zheng & al.[53]. Ils proposent de regrouper les root causes suivant leur lien au logiciel testé, aux cas de test ou à l'environnement.

7.1.1 Catégories liées au logiciel testé

1. **Async Wait :** Arrive quand un appel asynchrone est fait et n'attend pas la réponse pour continuer.[25]
2. **Concurrency :** Arrive quand plusieurs threads ont besoin des mêmes informations et que la gestion du verrouillage n'est pas gérée correctement.[25]
3. **Ressource leak :** Arrive quand l'application ne gère pas correctement ses ressources, ce qui peut entraîner un manque de ressources.[25]
4. **IO :**Une mauvaise gestion des canaux inputs ou outputs entraîne une erreur dans la génération de la fonctionnalité.[25]

5. **Randomness :** Utilisation d'un générateur de valeurs aléatoires sans avoir réfléchi à l'ensemble des nombres pouvant être générés.[25]

6. **Floating point operations :** Mauvaise gestion des types qui peuvent entraîner une trop grande ou trop faible précision.[25]

7. **Unordered collections :** La gestion des collections non ordonnées doit être faite en tant que telle. Si le code suppose le contraire, il pourrait y avoir du non déterminisme.[25]

8. **UI :** Mauvaise gestion de l'interface utilisateur qui peut entraîner des problèmes d'affichage bloquant l'application.[47]

9. **Program Logic :** La logique du développeur n'est pas la bonne et entraîne une erreur. Principalement lié à l'utilisation d'exception, ce qui explique qu'elle ne soit pas automatique.[47]

10. **NIO (Non-idempotent-outcome) :** Pour savoir si le test fait partie de cette catégorie, il faut toujours le faire fonctionner deux fois dans la même machine virtuelle de java et si le premier passe et le second est en erreur alors on dit qu'il "s'auto-pollue". [50]

7.1.2 Catégories liées aux cas de tests

1. **Test Order Dependency:** Si un test a besoin de valeurs venant d'un autre test, il y a une dépendance qui est faite au niveau de l'ordre. Le non déterminisme arrive si le test précédent ne s'est pas déroulé comme prévu.[25]

2. **Too restrictive range :** Certaines valeurs sont hors de la plage de possibilités d'un tableau par exemple et le test échoue quand le programme essaie de faire appel à elles. [12]

3. **Test case timeout :** Si un test grandit et qu'on n'augmente pas son temps d'exécution, celui ci peut arriver à dépasser ce temps et faire une erreur.[12]

4. **Test suite timeout :** Une suite de tests a un temps d'exécution maximale mais celle- ci n'est pas toujours ajustée correctement, par rapport à la quantité de tests et à la durée que cela représente.[12]

7.1.3 Catégories liées à l'environnement

1. **Network :** Le réseau est un lien pouvant être instable. Une erreur liée au réseau n'est pas toujours un signe d'erreur dans le code ou dans le test. Cependant, si il met le test en erreur, ça sous entend que le développeur n'a pas géré cette éventualité et doit en tenir compte.[25]

2. **Time :** Le temps est une variable complexe à utiliser. Si l'application a une visée internationale, il faudra bien tenir compte des possibles changements de fuseaux horaires. Actuellement, il faut aussi prendre en compte les heures d'été ou d'hiver ... ces variables peuvent entraîner une erreur indéterminée.[25]

3. **Platform Dependency :** L'erreur vient spécifiquement avec une plate-forme bien définie. C'est un flaky test qui peut apparaître facilement avec des applications car elles peuvent se trouver sur plusieurs types de plate-formes différentes.[12]

7.2 Identifier sa position dans le code

Pour aider les développeurs à corriger ces différentes root causes, Ziftci & al.[54] ont proposé un outil permettant de l'identifier directement dans le code où se trouve la divergence entre un test réussi et un test raté.

Cet outil va, lors de l'exécution des différents tests, garder en mémoire le moment où l'erreur survient. Cela va permettre de comparer avec une exécution réussie et ainsi mettre en avant, les lignes pouvant poser problème. D'après l'article et les retours faits par les différents testeurs, cet outil est très intéressant et peut vraiment aider les développeurs à gagner du temps, en leurs montrant précisément la ligne litigieuse.

Cet outil a quand même des limites qui sont à prendre en compte lors de son utilisation :

- **Limite de temps d'exécution :** Un temps limite d'exécution est fixé. Cela permet d'éviter que les traces d'exécution ne soient trop grandes.
- **Limite de taille :** Une limite sur le poids total des traces pour éviter d'utiliser trop de ressources.
- **Limite de langage utilisé :** Cet outil ne supporte que le C++ et le Java.
- **Limite de score de flakiness :** Le score de flakiness permet de s'assurer que le test échouera durant les différents tests.

Au vu des retours faits par les différents testeurs, cet outil a un intérêt certain. Il peut par contre, être amélioré au vu de son efficacité, en proposant une correction automatique sur des root causes précises.

7.3 Outils de classification

7.3.1 Execution clusters

Terragni & al.[46] ont proposé une idée pour classifier automatiquement les flaky tests rencontrés. Pour cela, ils ont proposé de travailler avec des "execution clusters", ce sont

des environnements virtuels qui explorent spécifiquement une exécution non-déterminée. Chacun va s'exécuter un nombre de fois délimité.

Dans leur article, ils ont proposé une liste de root-causes pouvant être classifiées grâce à leur nouvelle méthode. Ils ont sélectionné les root-causes les plus récurrentes en se basant sur la liste de Luo & al.[25]. Leur proposition, n'est pas définitive. Ils pensent qu'il y a moyen de créer de nouveaux clusters pouvant classifier d'autres root-causes.

Voici la liste :

1. Multi-threaded execution cluster.
2. Network execution cluster.
3. I/O execution cluster.
4. Test order execution cluster.
5. Platform cluster.

7.3.2 Torch Instrumentation

En collaboration avec Microsoft, Lam & al.[22] ont proposé un framework travaillant avec des fichiers .Nets et permettant de récolter de nombreuses informations pour détecter les flaky tests. Ce framewok, se nomme Torch Instrumentation et vient avec des plugins pour le profilage, la journalisation, l'injection de fautes, les tests de concurrence, le fuzzing de la planification des threads, etc. Une possibilité de modification a été pensée, car les utilisateurs peuvent ajouter des plugins, voir même en développer de nouveaux.

Torch fonctionne en modifiant tous les appels aux différentes API. Pour cela, il les remplace par des proxy qui pourront faire des appels à d'autres fonctions avant ou après l'appel de l'API en tant que tels. Pour cela, il existe des fonctions de Torch telle que OnStart qui seront appelées directement, avant d'appeler l'API original, OnEnd qui sera appelé après l'API original et OnException qui sera appelé si l'API renvoie une exception. OnStart renvoie un contexte qui est utilisé pour lier les informations venant des différentes fonctions.

Pour arriver à détecter les flaky tests, Torch utilise différents plugins de surveillance et de sauvegarde d'informations. Les chercheurs se sont rendus compte que les API se comportaient différemment suivant le résultat de l'exécution du processus. C'est pourquoi, il est important de recueillir toutes les analyses possibles pour arriver à comprendre ce qui rend le test instable.

Les informations récoltées sont :

- Les informations liées à l'API.

- Le temps de chaque appel.
- Les valeurs renvoyées par les fonctions OnEnd et OnException.
- L'id unique de l'objet reçu qui permet d'identifier les API travaillant sur le même objet.
- Les id des processus et des threads.
- L'id du thread parent. Cette information sert à connaître les dépendances des différents threads.

Les chercheurs avaient dans l'idée de réduire au maximum l'impact de cette prise d'information sur le temps de travail et les ressources nécessaires. C'est pourquoi, ils ont décidé de mettre en place des méthodes pour limiter la demande de ressources. D'un côté, ils passent les propriétés durant l'appel à OnStart. De l'autre, ils ont compressé les informations et ont décidé de les écrire de manière asynchrone sur le disque.

7.3.3 RootFinder

RootFinder est un outil permettant d'identifier les potentielles causes principales des flaky tests. Pour cela, il est utilisé en collaboration avec "Torch instrumentation" qui va lui fournir toutes les informations nécessaires à son bon fonctionnement.

Pour arriver à faire des propositions, RootFinder va passer sur chacun des fichiers journaux et les tester sur chaque prédicat implémenté dans son système.

Un prédicat est un booléen qui évalue l'état de la méthode à un moment donné. Dans l'étude proposée, il y avait six prédicats implémentés.

1. **Relative :** C'est utile pour identifier si une méthode non-déterministe retourne la même valeur lors d'un appel réussi.
2. **Absolute :** Ce prédicat est utile pour vérifier si la méthode retourne un code d'erreur ou une valeur null.
3. **Exception :** Permet de savoir si la méthode renvoie une exception.
4. **Order :** Ce prédicat sert à identifier les entrelacements de thread.
5. **Slow :** Vérifie si le temps pris pour l'exécution est plus long qu'un temps donné.
6. **Fast :** Vérifie si le temps pris pour l'exécution est plus court qu'un temps donné.

Quand il a fini de passer sur chacun des fichiers journaux, l'étape suivante va être de vérifier si certaines informations peuvent être tirées du travail précédent. Par exemple, un prédicat est toujours vrai ou toujours faux ou encore, varie. Il termine par les trier

pour donner une indication aux développeurs de ce qui est le plus probable comme cause de flaky test.

En fin de compte, RootFinder pourrait permettre d'identifier neuf des dix catégories proposées par Luo & al.[25].

7.4 Ce qu'il faut en retenir

Au final, la root cause des flaky tests est un point important dans la gestion de ceux-ci. Une connaissance approfondie des différentes root causes permet une meilleure compréhension de leurs fonctionnements et donc de la manière de les gérer. Cela permet entre autres, d'arriver à détecter plus rapidement les flaky tests mais aussi de localiser précisément la cause de flakiness dans le code. Ces possibilités ouvrent la porte à la correction automatique des tests.

Les classifications proposées par Luo & al.[25] sont communément acceptées. Cela permet d'avoir un vocabulaire et des termes qui sont de vrais points d'ancrages lors de la découverte mais aussi de discussions autour des flaky tests. Cela permet de différentier les causes liées aux logiciels, aux tests et à l'environnement.

La classification des flaky tests a été une des premières voies suivies par les chercheurs. De ce fait, des propositions d'outils ont été réalisées et ont pu être testées lors d'études approfondies. Ces outils peuvent aider les développeurs lors de la correction des tests. Ils pourront gagner du temps et avoir rapidement une meilleure compréhension du test.

8 Techniques de reproduction d'erreurs

Quand un logiciel ou un test ne produit pas ce qui est attendu, on appelle cela une erreur. Cela peut être très gênant dans l'utilisation du logiciel, car une erreur peut en entraîner d'autres, des modifications de données indésirables ou même l'arrêt complet du système si ça n'a pas été géré correctement.

C'est pour contrer ces aléas que les développeurs passent énormément de temps à essayer de traquer et corriger les erreurs. Quand elles sont récurrentes, il est assez facile pour le développeur d'arriver à la reproduire, de comprendre sa cause et de la corriger. Cependant, il se peut qu'une erreur n'arrive pas à chaque fois que l'on utilise le logiciel ou que l'on teste la fonctionnalité. Quand cela arrive, il devient très compliqué pour le développeur de comprendre la cause de cette erreur et de la corriger.

Arriver à reproduire les erreurs des flaky tests peut être très compliqué. Il y a énormément de facteurs pouvant jouer dans la cause de l'erreur. Le réseau, l'infrastructure, la charge de travail, le délai de réponses ... vous l'aurez compris, toutes ces causes sont les root causes présentées précédemment. Ces root causes peuvent être prises individuellement, mais peuvent aussi être liées ensemble pour donner la nature de l'erreur.

Pour reproduire l'erreur, la méthode employée par les développeurs est la ré-exécution. Celle-ci a un coût certain et un pourcentage de réussite assez aléatoire. Il serait donc intéressant de mettre en place des techniques de reproduction d'erreurs automatiques. Ceci pour gagner du temps lors de la phase de découverte de l'erreur et de compréhension de celle-ci.

Plusieurs études ont déjà été faites sur cette question, d'autant plus qu'elle ne doit pas spécifiquement être liée aux flaky tests. La reproduction d'erreurs concerne autant les logiciels déjà mis en fonction que ceux en cours de développement. Nous allons présenter des études et outils proposant la reproduction d'erreurs avec diverses méthodes et pouvant être utilisés dans le cadre des flaky tests.

8.1 Méthodes proposées

8.1.1 Record-replay

Une méthode proposée dans la littérature actuelle est la méthode Record-replay. Elle propose d'enregistrer et de reproduire le fonctionnement des logiciels grâce à des outils matériels ou des logiciels permettant de stocker les données générées durant le fonctionnement du logiciel surveillé.

La reproduction a un bon taux de réussites et en fait la méthode la plus efficace pour obtenir des informations fiables sur les erreurs logicielles. Hélas, le coût de génération est assez important, car il doit travailler lors de l'exécution du logiciel en tant que tel.

Nous pouvons par exemple penser à ReCrash[3]. Lors d'un crash logiciel, celui-ci génère un ensemble de tests qui reproduisent le problème rencontré par le logiciel.

L'utilisateur n'a plus qu'à les envoyer aux développeurs avec le rapport pour qu'ils puissent trouver plus facilement la source de l'erreur.

Les tests créés par ReCrash sont déterministes, ce qui est tout le contraire de nos flaky tests et ils apportent donc la solution à ceux-ci. Hélas, cette méthode a deux points négatifs :

- Celle-ci est coûteuse en espace de stockage même si cela peut être réduit, si les tests sont générés et évalués localement.
- La surcharge d'utilisation de ressources est énorme. Dans leurs articles, Artzi & al.[3] ont comparé le temps d'exécution et la consommation de mémoire pour un même programme, la variation va de 13% à 60% ce qui n'est pas faisable dans la pratique. Par contre, ils proposent un autre mode le "Second chance" qui lui n'augmente qu'entre 0% et 1.7% ce qui est bien plus acceptable.

Sans une optimisation de ReCrash, il n'est pas envisageable d'utiliser cette méthode pour reproduire les erreurs dans les flaky tests. Ceux-ci n'étant pas déterministes et pouvant être liés au temps de travail du processus, il semble impensable qu'on pourrait l'activer à chaque génération des tests.

8.1.2 Reproduction à partir de la stack trace

Une autre méthode de reproduction d'erreurs est l'approche basée sur la stack trace cette méthode est un processus qui va essayer de reproduire l'erreur après qu'elle soit arrivée. Il existe d'autres méthodes fonctionnant après que l'erreur survienne, mais la stack trace a l'avantage de pouvoir reproduire des erreurs en orienté objet ce qui n'est pas le cas de toutes les méthodes.

Après un crash logiciel ou l'erreur dans un test, le développeur va analyser la stack trace et le rapport d'erreur, il va essayer d'en trouver la cause, la classe et la méthode.

La reproduction à partir de la stack trace amène des avantages certains par rapport au record-replay. Premièrement, comme elle fonctionne après l'erreur, elle n'augmente pas la charge de performance nécessaire. Le développeur n'utilise que les informations récupérables lors de l'erreur. Ensuite, il ne faut pas démarrer de logiciel avant ou pendant l'exécution du logiciel. N'importe quel utilisateur, même un client, pourra transmettre les informations nécessaires à la résolution de l'erreur.

STAR :

Dans l'optique d'utiliser cette méthode, Chen & al.[7] ont proposé un outil se nommant STAR.

La figure 12 représente les différentes phases suivies par STAR pour arriver à générer des tests permettant la reproduction des erreurs.

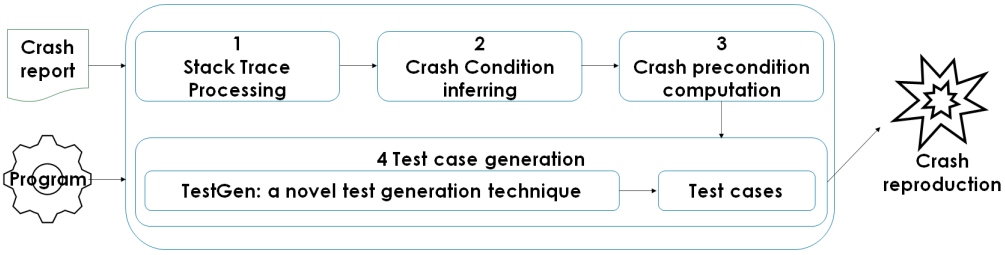


Figure 12: Architecture de STAR proposée par Chen & al.[7]

1. **Phase 1 :** STAR exécute le rapport d'erreur pour extraire les informations nécessaires à la suite de son fonctionnement. Il récupère l'id, le numéro de version et la stack trace de l'erreur.

2. **Phase 2 :** L'outil déduit grâce aux informations récoltées, les conditions de l'erreur et son emplacement dans le code. Lors de l'étude, STAR pouvait gérer trois types d'erreurs causées par trois exceptions.

   - Exception explicite : Ce type d'exception est déclenchée lorsque l'instruction de lancement est atteinte.
   - NullPointerException : Ce type d'exception est déclenchée quand une référence est équivalente à la valeur *null*. L'outil devra lire la référence pour déduire l'erreur.
   - ArrayIndexOutOfBoundsException : ce type d'exception est déclenchée quand l'index est hors des limites du tableau.

   Il se peut qu'il y ait plusieurs exceptions pouvant causer l'erreur, alors chaque exception sera utilisée pour créer les tests. Des développeurs pourraient, avec un peu de travail, améliorer le nombre d'exceptions supportées par STAR. Cependant, le positionnement de certaines exceptions sont difficiles à déduire.

3. **Phase 3 :** Avec les informations recueillies, la prochaine étape sert à essayer de déterminer comment déclencher l'erreur spécifique dans la méthode. Pour cela, ils ont adapté une approche basée sur l'exécution symbolique pour générer un ensemble de préconditions faibles pour arriver à activer l'erreur.

4. **Phase 4 :** Pour finir, STAR construit un test pour reproduire l'erreur cible avec une méthode visant à satisfaire la précondition générant l'erreur.

Au final, l'étude propose une réflexion sur l'outil et ses capacités. Il se montre utile pour les exceptions précitées avec un bon taux de reproduction. Mais de grands challenges sont toujours présents. Ils ont identifié les causes qui seront difficiles à résoudre telles que la dépendance à l'environnement qui du fait de sa relation externe avec le réseau ou des fichiers est difficile à reproduire. Ou encore les limitations du *SMT Solver* qui est utilisé pour valider les différentes propositions faites par l'outil.

Actuellement, l'outil n'est pas capable de résoudre des problèmes de concurrence ou de non-déterminisme, mais ça pourrait être un travail intéressant pour de nouvelles recherches.

RECORE :

STAR utilise les stack trace pour créer de nouveaux tests permettant de générer les erreurs obtenues précédemment. Pour faire cela, il déduit l'erreur grâce aux informations récoltées et aux pré-conditions et tente de créer le test générant l'erreur. RECORE proposé par Rößler & al.[39] propose une autre méthode pour générer un test arrivant à l'erreur visée.

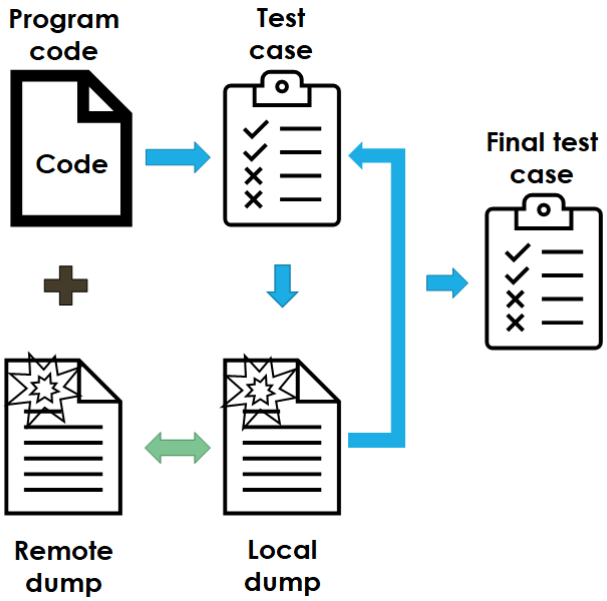


Figure 13: Fonctionnement de RECORE[39]

Pour commencer, RECORE a besoin de l'exécutable du programme (a) et du core dump de la machine ayant fait l'erreur (b). Il va ensuite utiliser EVOSUITE, qui, grâce aux informations fournies, va générer des tests (c). Ceux-ci feront eux-mêmes des core dump (d) qui seront comparés avec l'initial pour essayer de les rendre les plus similaires possible. Quand la trace originale aura pu être dupliquée alors RECORE proposera le test définitif (e).

Cette méthode ne vise pas spécifiquement des exceptions comme STAR. Elle fonctionne donc avec tous types d'erreur et c'est un de ses points forts. Un point d'attention doit quand même être fait, car à la sortie de l'étude en 2013 il ne gérait pas encore les programmes concurrents et ne pouvait donc pas proposer de test cohérent pour ce type d'erreur.

Botsing :

Plus récemment, en 2020, un nouvel outil du nom de Botsing a été proposé par Derakhshanfar & al.[10]. Cet outil open source est un framework de reproduction d'er-

reurs. Il utilise lui aussi la stack trace et l'exécutable de l'application pour générer des tests répliquant les erreurs. Par contre, ce qui le différencie des autres, c'est la possibilité d'utilisation du mécanisme de seeding avec l'utilisation de modèles ou des tests existants. Cette possibilité permet de proposer des tests pour des programmes orientés objet.

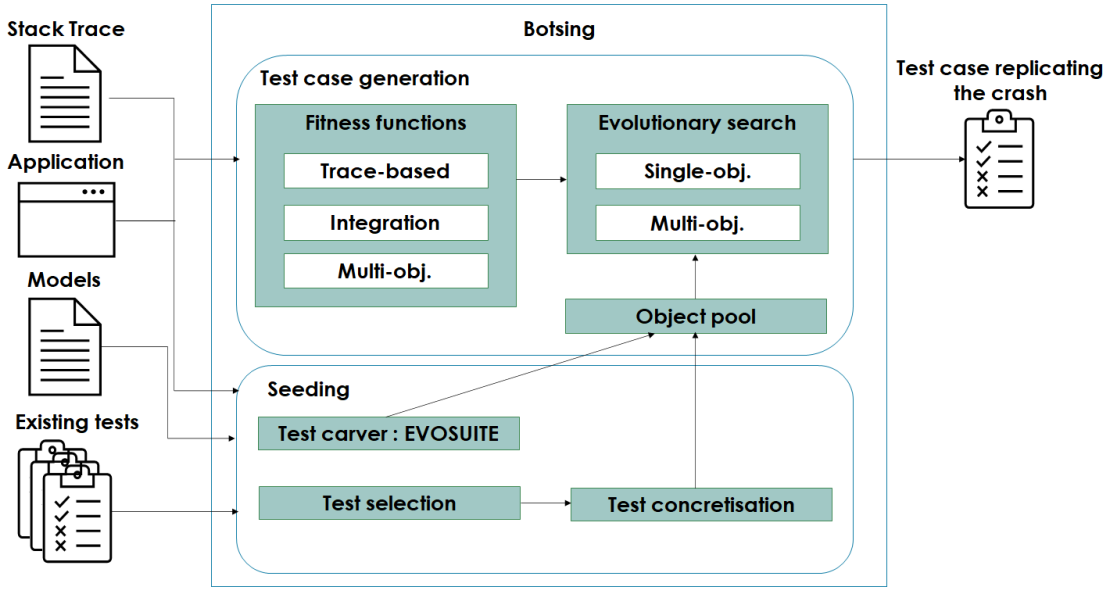


Figure 14: Architecture de Botsing[10]

La base de sa méthode de fonctionnement est similaire aux précédents modules proposés. Il récupère la stack trace Java ainsi que le fichier binaire de l'application. Cette stack trace va permettre d'indiquer le type d'exception précis ainsi qu'une liste de lignes de code du logiciel en lien avec l'exception. Ensuite, Botsing va cibler une ligne de la stack trace et sa classe associée. Les tests seront donc générés pour ce choix spécifique.

Suivant ce choix, la suite de son fonctionnement se fera sur base de l'*Evolutionary Search*. Botsing va générer plusieurs tests qui vont tenter d'activer l'erreur ciblée. Suite à cette génération , Botsing vérifie si un des tests arrive à reproduire la stack trace. Si ce n'est pas le cas, il fera une nouvelle itération de l'Evolution Search.

Après chaque itération, il va sélectionner le meilleur test pour générer les tests suivants. Pour le sélectionner, il va s'appuyer sur une des trois *Fitness Functions*[39, 45, 44] proposée de base. Celle-ci va mesurer la ressemblance entre le test proposé et le test à reproduire. Cela permettra de sélectionner le meilleur pour générer les suivants. Pour les générer, Botsing utilise deux opérations différentes : la mutation et le croisement. Ensuite, il vérifie que la méthode ciblée est toujours dans le test pour être certain de ne pas s'éloigner de l'objectif.

Pour les applications orientées objet, les valeurs des objets utilisés dans les tests, sont générés aléatoirement pour ne pas se limiter à une zone de recherche trop réduite. Hélas, cela ne garantit pas de rencontrer les pré-requis de spécifications. C'est pourquoi, Botsing propose d'utiliser un mécanisme de seeding.

Ce mécanisme de seeding se base sur deux stratégies différentes :

- **Test Seeding[37] :** Proposé par Rojas & al. ce mécanisme se base sur les tests déjà utilisés par l'application pour ses autres fonctionnalités. Le programme génère les tests et récupère les différents objets avec leurs valeurs pour les mettre dans une table et les utiliser ensuite, lors de la génération des nouveaux tests.
- **Behavioral Model Seeding[11] :** Ce mécanisme reçoit en entrée un ensemble de systèmes de transition représentant l'utilisation des diverses classes dans l'application. Des modèles sont ensuite créés pour exprimer les appels des fonctions dans les classes. Ces modèles sont ensuite utilisés pour créer les tests.

Lors de l'étude, Botsing arrivait à reproduire 66 des 124 erreurs proposées et 70 quand il utilisait le mécanisme de seeding. Ce framework est intéressant, car il a été développé avec l'objectif d'être extensible. Il peut donc être amélioré avec de nouvelles fonctionnalités ou algorithmes. Il pourrait être une proposition intéressante pour la reproduction d'erreurs et la correction de flaky tests.

8.1.3 Modification de test existants

Pour arriver à reproduire plus facilement une erreur, Xuan & al.[52] ont fait la proposition de prendre les tests déjà existants, de sélectionner ceux se trouvant en rapport avec la stack trace et de les modifier pour arriver à reproduire l'erreur rencontrée.

Pour ce faire, ils ont proposé un outil MuCrash. Celui-ci prend en entrée la stack trace de l'erreur, le source code et les tests existants. La sortie est un ensemble de tests modifiés qui peuvent reproduire l'erreur.

Il fonctionne en trois grandes étapes :

1. MuCrash exécute tous les tests existants et sélectionne ceux qui couvrent les classes présentes dans la stack trace.
2. MuCrash supprime toutes les assertions dans les tests sélectionnés en s'assurant du bon fonctionnement du test malgré tout.
3. MuCrash modifie les tests en changeant des opérateurs parmi une liste de possibilités (par exemple en mettant une variable à null ou en faisant un appel d'une méthode).

Chaque modification d'un test peut entraîner la création de plus d'un nouveau test. Chaque nouveau test créé et renvoyant l'erreur, est envoyé au développeur pour analyse manuelle.

Cette méthode se base principalement sur les tests déjà créés. Il faudra faire attention qu'une base de tests de bonne qualité et en quantité suffisante existe avant l'utilisation

de cette méthode. Un autre point à vérifier sera la sélection des opérateurs de mutation. Une sélection affinée de ces points permettra de réduire le nombre de cas générés et donc de réduire les tests sans valeurs.

8.1.4 Reproduction d'erreur de concurrence

Comme nous avons pu le voir dans la partie sur les root causes, la concurrence est une des premières causes d'erreur aléatoire. Par son utilisation de multiples threads, voir de multiples coeurs, la reproduction d'erreurs et sa résolution sont plus difficiles dans les programmes utilisant la concurrence.

C'est pourquoi, Weeratunge & al.[49] ont proposé une technique de reproduction d'erreurs pour les programmes concurrents. Le but est d'analyser l'erreur qui arrive dans un environnement multi-coeurs, de récupérer les différents journaux d'informations et de créer une reproduction pour un environnement mono-coeur. Cette reproduction sera plus simple à analyser par les développeurs, car ils pourront trouver plus facilement la source de l'erreur.

Cette méthode est proposée en collaboration avec un algorithme qui est chargé d'identifier le point de défaillance dans l'exécution. Grâce à cela, ils comparent les journaux d'erreurs entre le programme traité et le programme proposé pour être certains que l'erreur obtenue est bien similaire à celle ayant besoin d'être corrigée.

Leurs méthodes combinent une analyse légère d'un journal d'erreurs de processeur avec un algorithme qui utilise le résultat de l'analyse pour construire un programme qui pourra reproduire l'erreur dans un système mono-coeur. Cette façon de travailler permet de limiter un maximum la surcharge de travail lors de l'exécution du programme.

8.2 Ce qu'il faut en retenir

Nous avons présenté quatre méthodes différentes pour arriver à reproduire des erreurs. Ces méthodes ont des coûts et des fonctionnalités diverses et c'est pourquoi, il nous a semblé intéressant de proposer une vision ainsi qu'une explication de chacune d'elles. Dans le but d'aider les développeurs à faire le bon choix. Celui-ci devant être fait en conscience des avantages et inconvénients de chacune des méthodes.

- **Le record-replay :** L'avantage principal de cette méthode est son taux de réussite. Hélas, il présente des inconvénients importants. Il doit fonctionner en même temps que le logiciel et augmente de façon conséquente les ressources utilisées. Au vu de la difficulté de reproduction de certains flaky tests, ce n'est pas une solution acceptable pour arriver à les reproduire.
- **La reproduction à partir des stack trace :** Cette méthode est déjà plus intéressante dans notre cas. Pour des root causes précises, il suffit d'avoir la stack trace pour arriver à reproduire l'erreur. Bien que le taux de réussite soit plus

faible, les nombreux outils existants ont démontré leurs qualités. L'évolution est quand même nécessaire pour arriver à gérer automatiquement les nombreuses root causes pouvant survenir.

- **La modification des tests existants :** Cette approche travaille principalement sur les tests réalisés par les développeurs et tente de les modifier grâce à la stack trace. Les développeurs devront faire des tests de qualité pour que l'outil puisse proposer des modifications pouvant être utiles. Le taux de réussite présenté dans l'étude était correct aussi.
- **La reproduction d'erreur de concurrence :** Cette méthode est plus spécifique à une root cause, elle permet cependant de gérer des erreurs complexes car travaillant sur de multiples coeurs et processeurs. Elle permet de supprimer la concurrence en ne travaillant plus que sur un seul thread. Ce qui aidera grandement le développeur.

Toutes ces méthodes ont des avantages et des inconvénients. Dans le futur, on peut espérer qu'elles pourront travailler sur un nombre de root causes plus important et ainsi se présenter comme une solution fiable pour la reproduction d'erreurs.

9 Challenge avec des domaines parallèles

De plus en plus d'efforts ont été portés dans le domaine des flaky tests par la communauté scientifique. Ces efforts ont tout d'abord eu comme objectif de définir les root causes et leurs caractéristiques pour arriver à les détecter. Ensuite, ils se sont dirigés vers des méthodes de corrections automatiques. Récemment, certains chercheurs, ont décidé d'élargir le domaine en menant des recherches sur des domaines parallèles.

Dans ces domaines parallèles, les plate-formes telles qu'Android, Apple, Firefox, Chrome ... sont des domaines de choix car ils ne sont pas épargnés par les flaky tests et sont très utilisés, ce qui permet aux chercheurs d'avoir beaucoup de documentation disponible.

Les User Interface (UI) sont aussi des domaines pouvant bénéficier d'une meilleure connaissance des flaky tests les impactant. Ce domaine utilise un paramètre apportant beaucoup de non-déterminisme. Ce paramètre, c'est l'utilisateur. Le système est en attente constante d'évènements et d'informations fournies par l'utilisateur ce qui peut entraîner de l'indéterminisme et de l'instabilité.

Le dernier challenge qui sera présenté ici est en lien avec l'intégration continue. Celui-ci est en contact constant avec les tests et subit, dans de grandes mesures, les flaky tests. L'impact d'un flaky test sur l'intégration continue est important car elle nuit au bon déroulement de l'ensemble des tests mais aussi sur la confiance du développeur envers ces dits tests.

9.1 Challenge avec la technologie Android

Dans son étude, Thorve & al.[47] ont désigné des root causes spécifiques aux applications Android. Ils ont soutenu qu'il était important de prendre en compte ces catégories pour trois raisons :

1. **Les variations entre les plates-formes :** Il est bien connu qu'Android a eu une vitesse d'évolution assez rapide et nombre d'appareils utilisent encore d'anciennes versions. Il n'est pas possible pour les développeurs de tester chaque version spécifiquement ainsi que chaque plate-forme. De ce fait, il se peut donc que l'application soit instable.

2. **Les multiples connexions :** Comme dit précédemment, une application doit pouvoir tourner sur de nombreuses plate-formes diverses. Pour cela, il y a un nombre important de librairies différentes à utiliser. C'est pour ces raisons qu'il y a un risque augmenté de rencontrer de l'instabilité dans l'utilisation de l'application.

3. **La facilité de correction et de compréhension :** Une application Android est un élément simple avec des frontières précises. Ces fonctionnalités doivent être rapides à comprendre. La création d'outils permettant de corriger des flaky tests devrait être plus aisée qu'avec un logiciel installé sur un ordinateur ou un serveur.

9.1.1 Root cause

Durant l'étude, ils ont identifié cinq grandes catégories : la concurrence, la dépendance, la logique de programme, le réseau et l'interface utilisateur. Ils ont donc mis en valeur deux nouvelles catégories qui sont la logique de programme et l'interface utilisateur. Ces catégories sont liées à la qualité du programme en tant que tel et à l'interface utilisateur qui, si elle n'est pas pensée assez précisément, peut entraîner des erreurs.

9.1.2 Méthode de correction

Les développeurs Android ne sont pas très différents des autres développeurs. Modification du test pour le rendre le plus fiable possible, changement des bibliothèques utilisées pour employer celles qui n'ont pas le problème rencontré, correction de la logique utilisée, voire même, mise en commentaire du test pour qu'il n'apparaisse plus, sont toutes des méthodes appliquées dans les autres technologies.

9.2 Challenge avec les interfaces utilisateurs

L'étude de Romano & al.[38] s'est penchée sur la question des tests UI. Ils ont été, jusque-là, ignorés par les principaux travaux en lien avec les flaky tests déjà réalisés. Pourtant ces tests sont, la plupart du temps, plus complexes dûs à un nombre plus élevé de conditions d'exécution ainsi qu'une quantité de valeurs d'entrée plus grande. De plus, ces tests sont plus gourmands en ressources, ce qui les rend inadéquats à l'utilisation de la méthode de ré-exécutions.

Comparés aux tests traditionnels, les tests UI sont bien différents. Le nombre d'évènements liés aux interactions utilisateur, aux appels des différentes API, aux téléchargements et à l'affichage de multiples ressources sont hautement asynchrones par nature. Cela apporte de l'indéterminisme car déclenchés dans un ordre aléatoire. Ces flaky tests sont aussi plus compliqués à reproduire car il est difficile de penser à tous les scénarios possibles. L'utilisateur ayant un panel très large d'action.

Cette étude empirique souhaite identifier les root causes des flaky tests sélectionnés ainsi qu'une stratégie pour les reproduire et y remédier. Cela fournira une base pour pour d'autres chercheurs voulant proposer le développement de techniques de détection et de prévention.

9.2.1 Root cause

L'article présente quatre catégories principales de root causes :

1. **Attente asynchrone :** 45% des flaky tests analysés étaient de cette catégorie, la cause principale de cette erreur est liée à une mauvaise récupération de l'exécution des actions sur les objets ou les éléments de l'UI. Il en résulte une exception.

2. **Environnement :** Ces erreurs sont habituellement liées aux différences entre la plate-forme de test et l'application en tant que telle. Mais aussi entre la plate-forme Web et Android, iOS, etc... pour la plate-forme mobile.

3. **Problèmes d'API du générateur de test :** Cette catégorie est liée à l'utilisation d'API fournie par le framework de test. Celle-ci crée une erreur dans les fonctionnalités ou modifie le retour attendu.

4. **Problèmes de logique du test :** Cette catégorie est liée à la logique du test en lui-même. Ces erreurs peuvent être liées à des données utilisées dans un test précédent, ou des ressources non chargées, etc...

9.2.2 Méthode de reproduction d'erreurs

Les chercheurs ont investigué les méthodes utilisées pour arriver à reproduire les erreurs dues à des flaky tests. Ils ont proposé quatre stratégies différentes :

1. **Problème spécifique à la plate-forme :** Certains tests ne sont en erreur qu'avec une plate-forme spécifique. L'auteur du rapport doit donc spécifier la plate-forme utilisée pour générer l'erreur.

2. **Réorganiser/Réduire la suite de test :** Une autre méthode de reproduction est la modification de l'ordre des tests. Certaines erreurs sont order-dépendantes et cette stratégie permet de la générer.

3. **Fournir un extrait de code :** L'utilisation de portions de code peut entraîner des erreurs. C'est pourquoi certains développeurs utilisent cette méthode pour arriver à les reproduire.

4. **Forcer les conditions d'environnement :** La dernière méthode proposée pour arriver à générer les erreurs est d'utiliser le test dans des conditions d'utilisations normales.

9.2.3 Méthode de correction

Pour finir, ils ont cherché les différentes méthodes de correction utilisées par les développeurs. Ils ont proposé quatre catégories :

1. **Délai :** Une des propositions est de modifier le délai entre l'action et le chargement des informations. Cela permettra d'éviter aux tests de continuer à s'exécuter sans avoir les informations nécessaires. La charge processeur peut entraîner un délai plus long suivant les demandes en cours. Augmenter ce délai offre du temps supplémentaire au système pour être prêt à fournir les informations nécessaires.

2. **Dépendance externe :** Certains tests ont pu être résolus en modifiant l'ordre d'appel des API ou en corrigeant l'appel en lui-même car pas utilisé de la bonne façon.

3. **Refonte des contrôles de test :** La logique d'un test peut parfois changer au fil du temps et doit être modifié pour que l'implémentation soit toujours correcte.

4. **Désactiver les fonctionnalités / Supprimer le test :** Cette méthode est la plus expéditive de toutes mais elle aura le mérite d'empêcher une erreur lors de la génération de la suite des tests.

Comme nous pouvons le voir, les propositions de reproduction et de correction sont assez rudimentaires. Ce domaine de recherche peut offrir un challenge intéressant pour les chercheurs. Les flaky tests sont nombreux lors de l'utilisation d'UI il est donc important de ne pas laisser ce domaine qui conduit à la bonne connaissance et gestion des flaky tests, sur le bord du chemin.

9.3 Challenge avec l'intégration continue

Les développeurs utilisent de plus en plus souvent un environnement d'intégration continue, celui-ci est chargé de tester automatiquement les fonctionnalités existantes et de vérifier qu'il n'y a pas de régression dans le code. Pour cela, les tests sont exécutés à chaque envoi de code et les résultats indiquent si le changement doit être intégré dans le code de production ou non.

Pour un projet de faible ampleur, ce mode de fonctionnement est parfaitement acceptable. Par contre, si l'ampleur du projet augmente, le nombre de tests augmentera et la charge de travail ainsi que son temps d'exécution en seront impactés. C'est pourquoi certains chercheurs ont proposé de mettre en place un RTS.

Un RTS sert à réduire le coût des tests de régression en sélectionnant seulement les tests qui sont affectés par un changement. Actuellement, l'industrie a adopté le "module-level RTS" pour leurs environnements CI, mais des chercheurs ont proposé de nouvelles méthodes qui se nomment "class-level RTS" ou bien un mode hybride reprenant les deux méthodes.

Lors de leurs recherches, Shi & al.[43] ont découvert en comparant la sélection faite par le RTS et la liste complète des tests, que le RTS avait exclu tous les tests en erreur. Ces tests sont presque tous des flaky tests.

Comme dit précédemment, l'intégration continue et les tests de régression sont fortement impactés par les flaky tests. Ils font perdre du temps de génération au système, ils peuvent impacter d'autres tests, la confiance des développeurs dans les résultats peut être ébranlée, etc... C'est pourquoi, un flaky test n'est pas le bienvenu dans les tests de régression. Ils n'apportent pas d'information fiable sur les changements apportés et ne correspondent pas aux fonctionnalités recherchées quand on utilise un environnement CI.

Bien que cette technique ne soit pas développée pour éviter les flaky tests, Shi & al. ont découvert empiriquement que le RTS le permet. Ils ont été les premiers à analyser la différence de résultat entre le RTS et la génération complète des tests. De ce fait, ils sont tombés par hasard sur cette fonctionnalité du RTS.

Cette information permet de rendre la technique RTS encore plus intéressante car elle permettra aux développeurs de gagner du temps mais aussi d'éviter de les induire en erreur avec des flaky tests. Cette nouvelle fonctionnalité pourrait ouvrir la voie à de nouvelles méthodes de détection ou de correction des flaky tests.

9.4 Ce qu'il faut en retenir

Le domaine des flaky tests a des ramifications dans de nombreux domaines. De nouvelles idées peuvent arriver de domaines qui ne sont pas à priori moteurs sur le sujet. Il est donc important pour les chercheurs de garder des liens entre les différents domaines ainsi qu'une curiosité pour les domaines annexes.

De nouvelles voies de recherche en lien avec les flaky tests sont encore actuellement inexplorées et de beaux challenges attendent les chercheurs. La démocratisation des flaky tests permettra une meilleure connaissance de ce terme, un emploi plus systématique et ainsi une remontée d'informations intéressantes pourrait arriver dans des domaines insoupçonnés.

10 Lien entre flaky tests et stratégies correctives

10.1 Nouveaux domaines

Lors des premières études sur le domaine, les propositions de lien entre flaky test et stratégies sont toujours assez basiques. Le domaine n'étant pas encore bien fixé, il n'y a pas de proposition de stratégies automatiques ou d'outils permettant une correction facilitée pour les développeurs. Les chercheurs proposent donc des solutions empiriques pour arriver à corriger les différents flaky tests.

Luo & al.[25] ont commencé en proposant des liens entre la catégorie de flaky test et le type de correction. Nous pouvons d'ailleurs voir dans la Figure 15 leurs propositions. Les diverses stratégies données sont des modifications à faire dans le code par le développeur. Bien que ces stratégies soient basiques, cela ne veut pas dire qu'elles sont mauvaises. Elles donnent des réponses aux questions des développeurs ce qui est l'objectif principal.

Causes	Stratégies
Async wait	Ajouter/modifier le waitFor Ajouter/modifier le sleep Changer l'ordre d'execution Autres
Concurrence	Verrouiller les opérations atomiques Rendre déterministe Changer les conditions Change les assertions Autres
Test dépendant de l'ordre	Configurer/nettoyer les états Supprimer les dépendances Fussionner les tests

Figure 15: Proposition de lien fait par Luo & al.[25]

Nous pouvons d'ailleurs comparer la technique de Luo & al. à celles faites dans l'analyse empirique sur les flaky tests, basées sur les interfaces utilisateurs, proposées par Romano & al.[38] ou bien encore dans l'étude de Thorve & al. [47] sur les flaky tests dans les applications Android.

Dans la Figure 16 nous pouvons voir les différentes propositions faites pour les interfaces utilisateurs et la Figure 17 nous présente les propositions faites pour Android. Chaque proposition recèle de stratégies ayant été utilisées par des développeurs pour gérer les flaky tests rencontrés dans leurs projets. Chaque tableau représente les différents types de flaky tests rencontrés et les stratégies utilisées pour les corriger.

10.2 Domaines expérimentés

Depuis l'article de Luo & al. en 2014, de nombreuses propositions ont été faites pour arriver à détecter, reproduire et corriger de manière plus automatisée les flaky tests.

Causes	Stratégies
Async wait	Désactiver les animations Ajouter du délais Corriger le méchanisme d'attente Repenser la logique
Test script Problème de logique	Corriger le méchanisme d'attente Repenser la logique
Problème d'API avec le gestionnaire de test	Corriger le méchanisme d'attente Repenser la logique Corriger l'accès aux API
Environnement	Repenser la logique Changer la version de la librairie

Figure 16: Proposition de lien fait par Romano & al.[38]

Causes	Stratégies
Concurrence	Améliorer l'implémentation Modifier les assertions
Dépendance	Améliorer l'implémentation Modifier l'implémentation Ré-exécuter Modifier les assertions Supprimer des tests
Logique du programme	Améliorer l'implémentation Modifier les assertions
Réseau	Améliorer l'implémentation Modifier l'implémentation
Interface utilisateur	Améliorer l'implémentation
Inclassifiable	Améliorer l'implémentation Ré-exécuter Supprimer des tests

Figure 17: Proposition de lien fait par Thorve & al.[47]

Différents travaux pour rassembler toutes ces propositions ont déjà été proposés par Zheng & al.[53] et Verdecchia & al.[48] en 2021.

L'étude de Zheng & al. propose de faire le point sur les progrès faits par la recherche sur les flaky tests. Ils présentent les principales causes des flaky tests ainsi que les propositions de stratégies et d'approches correctives faites par les chercheurs. L'étude s'est penchée sur un nombre important de techniques différentes pour arriver à détecter les flaky tests ainsi que leurs causes. Si nous comparons les études avec la section précédente, nous pouvons détecter une nette différence au niveau de la complexité des stratégies proposées. La maturité du domaine apporte une forte utilisation d'outils et stratégies faisant appel à des techniques n'utilisant pas la ré-exécution et inversement, des techniques de mutation de test flakiness et des appels asynchrones. Une bonne compréhension du domaine permet aux chercheurs de proposer des outils plus complexes avec un meilleur niveau d'automatisation de la correction ainsi qu'un bon taux de réussite.

De son côté, l'étude de Verdecchia & al. propose de présenter une nouvelle approche en la comparant avec des outils proposés précédemment et ainsi démontrer l'efficacité

de la nouvelle approche. Cette méthode permet aux chercheurs et aux développeurs de comparer facilement et rapidement chaque outil, connaître leurs avantages et inconvénients par rapport aux autres et ainsi pouvoir faire un choix plus aisé dans la méthode à suivre quand on est confronté à un type de flaky test bien précis.

Ces travaux ont un intérêt certain et permettent de guider les nouveaux entrants dans le domaine vers de nouvelles pistes de recherche.

11 Discussion

Voici la dernière partie de ce travail, il est temps de proposer une réponse aux différentes questions ayant conduit à l'écriture de celui-ci. De mettre en lien toutes ses parties et de faire émerger les idées, telles des briques, permettant de construire les réponses souhaitées.

11.1 Catégories de flaky tests

Au démarrage de ce travail, diverses questions se sont posées sur l'état de la recherche dans le domaine des flaky tests. Existait-il des catégories de flaky tests et si oui, des liens entre les techniques de debug et ces catégories étaient-ils déjà proposés ?

Grâce à la mapping study faite en amont, aux choix des critères d'inclusion et d'exclusion ainsi qu'à la sélection des articles, il a été possible de démontrer que des catégories ont déjà été proposées. D'ailleurs, diverses études ont été présentées dans ce sens. Parmi celles-ci, Luo & al.[25], ont proposé pour la première fois une liste de Root causes servant à différencier les flaky tests qui ont pu être rencontrés. Pour parvenir à cela, ils ont dû analyser 201 commits dans 51 projets open-source. Ce qui a permis d'obtenir une liste de dix catégories différentes de flaky tests.

Ce ne sont pas les seuls, plusieurs autres chercheurs ont proposé d'affiner cette proposition en ajoutant soit de nouvelles catégories, ce qui a été fait par Thorve & al.[47], Wei & al.[50] ou bien encore Eck & al.[12], en regroupant les root causes par thèmes. Ce qui a été énoncé par Zheng & al.[53]

Il est d'ailleurs intéressant de préciser que des études ont proposé d'automatiser cette catégorisation. Des outils et méthodes ont été proposés pour classifier les flaky tests et arriver à identifier dans le code, leur source de flakiness.

Bien que RootFinder identifie neuf des dix catégories proposées par Luo & al., les outils proposés ne classifient pas encore tous les types de root causes connus. Ces travaux devront être améliorés pour réduire le nombre d'outils nécessaires à la bonne gestion des flaky tests.

11.2 Techniques de debug

Pour ce qui est du lien entre techniques de debug et catégories, certaines études ont déjà fait des propositions. Une section de ce travail a d'ailleurs abordé ce sujet. Hélas, les propositions sont souvent limitées à la détection des flaky tests ou aux stratégies de correction. De nombreuses études proposent des solutions ciblées à une catégorie ou à une root cause précise. C'est plus aisé à expliquer et à démontrer mais cela augmente le nombre d'outils à déployer si l'ont veut les mettre tous, en pratique.

Il serait intéressant de proposer une étude rassemblant l'ensemble des thèmes proposés dans ce travail. La prédiction, la détection, la classification avec l'identification

de la source de flakiness, les stratégies de correction et de reproduction d'erreurs sont toutes des branches de recherche sur les flaky tests ne demandant qu'à être regroupées.

11.3 Travaux futurs

Pour le futur, il semble important d'écouter les besoins des développeurs étant en contact avec les flaky tests. Des études ont été menées récemment dans cet optique et une liste d'envies a pu en être extraite. Il semble qu'une intégration des outils dans les IDE est fortement demandée, mais il semble tout aussi important de former les nouveaux développeurs à cette problématique. Un guide décrivant les flaky tests et leurs conséquences devraient émerger pour éveiller les consciences sur cette nouvelle problématique. Cela permettrait de mettre en place de meilleurs pratiques permettant notamment d'éviter l'apparition de flaky tests.

12 Conclusion

Ce travail démontre, qu'au fil des ans, une avancée significative du domaine a été faite au niveau de la classification des flaky tests et de leurs méthodes de détection. De nombreuses études se sont penchées sur ce sujet et ont apporté leurs savoirs pour améliorer les connaissances collectives.

Grâce à cela, nous pouvons clairement répondre qu'il existe bel et bien des catégories de flaky tests identifiées et bien connues des chercheurs. De nombreuses études se basent sur ces catégories communément acceptées pour proposer des outils et des stratégies de gestion des flaky tests.

Cependant, bien que des études suggèrent des propositions de lien entre stratégies et catégories de flaky tests, nous ne pouvons pas affirmer que ce travail soit fini. De nombreuses méthodes dans les articles cités n'ont pas été explorées. Telles que la mutation des tests ou la reproduction à partir de la stack trace. Des thèmes de recherche sont encore actuellement peut développés dans la littérature. La plupart des méthodes de reproduction d'erreurs ne sont pas spécifiquement liées aux flaky tests. Alors que ce lien permettrait d'aider les méthodes de détection et de correction. Le fait que les chercheurs n'aient pas encore développé la recherche dans cette direction est une surprise.

Il ne faut pas oublier que l'objectif du domaine n'est pas uniquement de découvrir des outils ou stratégies permettant de détecter et corriger les flaky tests, l'objectif est aussi de réduire pro-activement la génération de tels tests en prédisant les méthodes ou le vocabulaire pouvant apporter de l'instabilité dans les tests. L'objectif est aussi d'arriver à réduire le coût de ré-exécution des erreurs en reproduisant plus facilement les flaky tests.

À la lumière de toutes ces considérations, une étude proposant un document liant des méthodes acceptées par le plus grand nombre, pour chaque catégorie, serait un apport important pour le domaine.

References

[1] Azeem Ahmad, Ola Leifler, and Kristian Sandahl. Empirical analysis of factors and their effect on test flakiness-practitioners' perceptions. arXiv preprint arXiv:1906.00673, 2019.

[2] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. Flakeflagger: Predicting flakiness without rerunning tests. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pages 1572–1584. IEEE, 2021.

[3] Shay Artzi, Sunghun Kim, and Michael D Ernst. Recrash: Making software failures reproducible by preserving object states. In European conference on object-oriented programming, pages 542–565. Springer, 2008.

[4] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. Deflaker: Automatically detecting flaky tests. In 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), pages 433–444. IEEE, 2018.

[5] Gregor V Bochmann and Alexandre Petrenko. Protocol testing: review of methods and relevance for software testing. In Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis, pages 109–124, 1994.

[6] Bruno Henrique Pachulski Camara, Marco Aurélio Graciotto Silva, Andre T Endo, and Silvia Regina Vergilio. What is the vocabulary of flaky tests? an extended replication. In 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), pages 444–454. IEEE, 2021.

[7] Ning Chen and Sunghun Kim. Star: Stack trace based automatic crash reproduction via symbolic execution. IEEE transactions on software engineering, 41(2):198–220, 2014.

[8] TIBCO CLOUD. Integration continue. https://www.tibco.com/fr/reference-center/what-is-continuous-integration.

[9] Juan Cruz-Benito. Systematic literature review & mapping, November 2016. I would like to thank the European Social Fund and the Consejería de Educación of the Junta de Castilla y León (Spain) for funding my pre- doctoral fellow contract.

[10] Pouria Derakhshanfar, Xavier Devroey, Annibale Panichella, Andy Zaidman, and Arie Van Deursen. Botsing, a search-based crash reproduction framework for java. In 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 1278–1282. IEEE, 2020.

[11] Pouria Derakhshanfar, Xavier Devroey, Gilles Perrouin, Andy Zaidman, and Arie van Deursen. Search-based crash reproduction using behavioural model seeding. Software Testing, Verification and Reliability, 30(3):e1733, 2020.

[12] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. Understanding flaky tests: The developer's perspective. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 830–840, 2019.

[13] Mark Fewster and Dorothy Graham. Software test automation. Addison-Wesley Reading, 1999.

[14] Martin Fowler and Matthew Foemmel. Continuous integration, 2006.

[15] Gordon Fraser and Andrea Arcuri. Whole test suite generation. IEEE Transactions on Software Engineering, 39(2):276–291, 2012.

[16] Yoav Freund and Llew Mason. The alternating decision tree learning algorithm. In icml, volume 99, pages 124–133, 1999.

[17] Sarra Habchi, Guillaume Haben, Mike Papadakis, Maxime Cordy, and Yves Le Traon. A qualitative study on the sources, impacts, and mitigation strategies of flaky tests. In 2022 IEEE Conference on Software Testing, Verification and Validation (ICST), pages 244–255. IEEE, 2022.

[18] Guillaume Haben, Sarra Habchi, Mike Papadakis, Maxime Cordy, and Yves Le Traon. A replication study on the usability of code vocabulary in predicting flaky tests. In 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), pages 219–229. IEEE, 2021.

[19] Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification. IBM Systems Journal, 41(1):4–12, 2002.

[20] Muhammad Abid Jamil, Muhammad Arif, Normi Sham Awang Abubakar, and Akhlaq Ahmad. Software testing techniques: A literature review. In 2016 6th international conference on information and communication technology for the Muslim world (ICT4M), pages 177–182. IEEE, 2016.

[21] Barbara A Kitchenham. Systematic review in software engineering: where we are and where we should be going. In Proceedings of the 2nd international workshop on Evidential assessment of software technologies, pages 1–2, 2012.

[22] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. Root causing flaky tests in a large-scale industrial setting. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 101–111, 2019.

[23] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. idflakies: A framework for detecting and partially classifying flaky tests. In 2019 12th ieee conference on software testing, validation and verification (icst), pages 312–322. IEEE, 2019.

[24] Wing Lam, Stefan Winter, Angello Astorga, Victoria Stodden, and Darko Marinov. Understanding reproducibility and characteristics of flaky tests through test reruns in java projects. In 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE), pages 403–413. IEEE, 2020.

[25] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, pages 643–653, 2014.

[26] William S Noble. What is a support vector machine? Nature biotechnology, 24(12):1565–1567, 2006.

[27] Oracle. Javadoc - hashset. https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/HashSet.html.

[28] M Paegelow and LUCC Budget. In geomatic approaches for modeling land change scenarios. Camacho Olmedo, MT, Paegelow, M., Mas, JF, Escobar, F., Eds, pages 437–440, 2018.

[29] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. Flake it'till you make it: Using automated repair to induce and fix latent test flakiness. In Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, pages 11–12, 2020.

[30] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. Surveying the developer experience of flaky tests. In Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2022.

[31] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. Systematic mapping studies in software engineering. In 12th International Conference on Evaluation and Assessment in Software Engineering (EASE) 12, pages 1–10, 2008.

[32] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. Information and software technology, 64:1–18, 2015.

[33] Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo d'Amorim, Christoph Treude, and Antonia Bertolino. What is the vocabulary of flaky tests? In Proceedings of the 17th International Conference on Mining Software Repositories, pages 492–502, 2020.

[34] Valeria Pontillo. Static test flakiness prediction. In Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, page 325–327, 2022.

[35] Valeria Pontillo, Fabio Palomba, and Filomena Ferrucci. Toward static test flakiness prediction: A feasibility study. In Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution, pages 19–24, 2021.

[36] Md Tajmilur Rahman and Peter C Rigby. The impact of failing, flaky, and high failure tests on the number of crash reports associated with firefox builds. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 857–862, 2018.

[37] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. Seeding strategies in search-based unit test generation. Software Testing, Verification and Reliability, 26(5):366–401, 2016.

[38] Alan Romano, Zihe Song, Sampath Grandhi, Wei Yang, and Weihang Wang. An empirical analysis of ui-based flaky tests. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pages 1585–1597. IEEE, 2021.

[39] Jeremias Rößler, Andreas Zeller, Gordon Fraser, Cristian Zamfir, and George Candea. Reconstructing core dumps. In 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, pages 114–123. IEEE, 2013.

[40] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. IEEE Transactions on software engineering, 22(8):529–551, 1996.

[41] Claude Sammut and Geoffrey I Webb. Encyclopedia of machine learning. Springer Science & Business Media, 2011.

[42] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. Detecting assumptions on deterministic implementations of non-deterministic specifications. In 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), pages 80–90. IEEE, 2016.

[43] August Shi, Peiyuan Zhao, and Darko Marinov. Understanding and improving regression test selection in continuous integration. In 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE), pages 228–238. IEEE, 2019.

[44] Mozhan Soltani, Pouria Derakhshanfar, Annibale Panichella, Xavier Devroey, Andy Zaidman, and Arie van Deursen. Single-objective versus multi-objectivized optimization for evolutionary crash reproduction. In International Symposium on Search Based Software Engineering, pages 325–340. Springer, 2018.

[45] Mozhan Soltani, Annibale Panichella, and Arie Van Deursen. Search-based crash reproduction and its impact on debugging. IEEE Transactions on Software Engineering, 46(12):1294–1317, 2018.

[46] Valerio Terragni, Pasquale Salza, and Filomena Ferrucci. A container-based infrastructure for fuzzy-driven root causing of flaky tests. In 2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER), pages 69–72. IEEE, 2020.

[47] Swapna Thorve, Chandani Sreshtha, and Na Meng. An empirical study of flaky tests in android apps. In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 534–538. IEEE, 2018.

[48] Roberto Verdecchia, Emilio Cruciani, Breno Miranda, and Antonia Bertolino. Know you neighbor: Fast static prediction of test flakiness. IEEE Access, 9:76119–76134, 2021.

[49] Dasarath Weeratunge, Xiangyu Zhang, and Suresh Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. In Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems, pages 155–166, 2010.

[50] Anjiang Wei, Pu Yi, Zhengxi Li, Tao Xie, Darko Marinov, and Wing Lam. Preempting flaky tests via non-idempotent-outcome tests. In International Conference on Software Engineering (ICSE'22), 2022.

[51] W Eric Wong, Joseph R Horgan, Saul London, and Hiralal Agrawal. A study of effective regression testing in practice. In PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering, pages 264–274. IEEE, 1997.

[52] Jifeng Xuan, Xiaoyuan Xie, and Martin Monperrus. Crash reproduction via test case mutation: Let existing test cases help. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pages 910–913, 2015.

[53] Wei Zheng, Guoliang Liu, Manqing Zhang, Xiang Chen, and Wenqiao Zhao. Research progress of flaky tests. In 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 639–646. IEEE, 2021.

[54] Celal Ziftci and Diego Cavalcanti. De-flake your tests: Automatically locating root causes of flaky tests in code at google. In 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 736–745. IEEE, 2020.

Annexe A - Tableau des références

Titre de l'article	Références
An empirical analysis of flaky tests	Luo, Q., Hariri, F., Eloussi, L., & Marinov, D. (2014, November). An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering* (pp. 643-653).
An empirical study of bugs in test code	Vahabzadeh, A., Fard, A. M., & Mesbah, A. (2015, September). An empirical study of bugs in test code. In *2015 IEEE international conference on software maintenance and evolution (ICSME)* (pp. 101-110). IEEE.
Detecting assumptions on deterministic implementations of non-deterministic specifications	Shi, A., Gyori, A., Legunsen, O., & Marinov, D. (2016, April). Detecting assumptions on deterministic implementations of non-deterministic specifications. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)* (pp. 80-90). IEEE.
Winning with Flaky Test Automation	Roseberry, W. M. (2016). Winning with Flaky Test Automation.
An Empirical Study of Flaky Tests in Android Apps	Thorve, S., Sreshtha, C., & Meng, N. (2018, September). An empirical study of flaky tests in android apps. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 534-538). IEEE.
DeFlaker: Automatically Detecting Flaky Tests	Bell, J., Legunsen, O., Hilton, M., Eloussi, L., Yung, T., & Marinov, D. (2018, May). DeFlaker: Automatically detecting flaky tests. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (pp. 433-444). IEEE.
Practical Test Dependency Detection	Gambi, A., Bell, J., & Zeller, A. (2018, April). Practical test dependency detection. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)* (pp. 1-11). IEEE.
The Impact of Failing, Flaky, and High Failure Tests on the Number of Crash Reports Associated with Firefox Builds	Rahman, M. T., & Rigby, P. C. (2018, October). The impact of failing, flaky, and high failure tests on the number of crash reports associated with Firefox builds. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 857-862).
Towards a Bayesian Network Model for Predicting Flaky Automated Tests	King, T. M., Santiago, D., Phillips, J., & Clarke, P. J. (2018, July). Towards a bayesian network model for predicting flaky automated tests. In 2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C) (pp. 100-107). IEEE.
Identifying key success factors in stopping flaky tests in automated REST service testing	Mascheroni, M. A., & Irrazábal, E. (2018). Identifying key success factors in stopping flaky tests in automated REST service testing. *Journal of Computer Science & Technology*, *18*.
Single-objective Versus Multi-objectivized Optimization for Evolutionary Crash Reproduction	Soltani, M., Derakhshanfar, P., Panichella, A., Devroey, X., Zaidman, A., & Deursen, A. V. (2018, September). Single-objective versus multi-objectivized optimization for evolutionary crash reproduction. In *International Symposium on Search Based Software Engineering* (pp. 325-340). Springer, Cham.
Empirical analysis of factors and their effect on test flakiness-practitioners' perceptions	Ahmad, A., Leifler, O., & Sandahl, K. (2019). Empirical analysis of factors and their effect on test flakiness-practitioners' perceptions. *arXiv preprint arXiv:1906.00673*.
iDFlakies: A framework for detecting and partially classifying flaky tests	Lam, W., Oei, R., Shi, A., Marinov, D., & Xie, T. (2019, April). iDFlakies: A framework for detecting and partially classifying flaky tests. In *2019 12th ieee conference on software testing, validation and verification (icst)* (pp. 312-322). IEEE.
iFixFlakies: a framework for automatically fixing order-d	Shi, A., Lam, W., Oei, R., Xie, T., & Marinov, D. (2019, August). iFixFlakies: A framework for automatically fixing order-dependent flaky tests. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 545-555).
Root causing flaky tests in a large-scale industrial setting	Lam, W., Godefroid, P., Nath, S., Santhiar, A., & Thummalapenta, S. (2019, July). Root causing flaky tests in a large-scale industrial setting. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 101-111).
Understanding and Improving Regression Test Selection in Continuous Integration	Shi, A., Zhao, P., & Marinov, D. (2019, October). Understanding and improving regression test selection in continuous integration. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)* (pp. 228-238). IEEE.
Understanding flaky tests: the developer's perspective	Eck, M., Palomba, F., Castelluccio, M., & Bacchelli, A. (2019, August). Understanding flaky tests: The developer's perspective. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 830-840).
An experimental study on flakiness and fragility of randoop regression test suites	Paydar, S., & Azamnouri, A. (2019, May). An experimental study on flakiness and fragility of randoop regression test suites. In *International Conference on Fundamentals of Software Engineering* (pp. 111-126). Springer, Cham.
Flaky Tests: Problems, Solutions, and Challenges.	Palomba, F. (2019). Flaky Tests: Problems, Solutions, and Challenges. In *BENEVOL*.
FlakyLoc flakiness localization for reliable test suites in web applications	Morán Barbón, J., Augusto Alonso, C., Bertolino, A., Riva Alvarez, C. A., & Tuya González, P. J. (2020). FlakyLoc: flakiness localization for reliable test suites in web applications. *Journal of Web Engineering*, *2*.
Mitigating the effects of flaky tests on mutation testing	Shi, A., Bell, J., & Marinov, D. (2019, July). Mitigating the effects of flaky tests on mutation testing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 112-122).
Predictive test selection	Machalica, M., Samylkin, A., Porth, M., & Chandra, S. (2019, May). Predictive test selection. In 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP) (pp. 91-100). IEEE.
A Container-Based Infrastructure for Fuzzy-Driven Root Causing of Flaky Tests	Terragni, V., Salza, P., & Ferrucci, F. (2020, October). A container-based infrastructure for fuzzy-driven root causing of flaky tests. In 2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER) (pp. 69-72). IEEE.

A large-scale longitudinal study of flaky tests	Lam, W., Winter, S., Wei, A., Xie, T., Marinov, D., & Bell, J. (2020). A large-scale longitudinal study of flaky tests. *Proceedings of the ACM on Programming Languages*, *4*(OOPSLA), 1-29.
A Study on the Lifecycle of Flaky Tests	Lam, W., Muşlu, K., Sajnani, H., & Thummalapenta, S. (2020, June). A study on the lifecycle of flaky tests. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (pp. 1471-1482).
Debugging Flaky Tests on Web Applications.	Morán, J., Augusto, C., Bertolino, A., de la Riva, C., & Tuya, J. (2019, September). Debugging Flaky Tests on Web Applications. In *WEBIST* (pp. 454-461).
De-flake your tests: Automatically locating root causes of flaky tests in code at google	Ziftci, C., & Cavalcanti, D. (2020, September). De-flake your tests: Automatically locating root causes of flaky tests in code at google. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 736-745). IEEE.
Dependent-test-aware regression testing techniques	Lam, W., Shi, A., Oei, R., Zhang, S., Ernst, M. D., & Xie, T. (2020, July). Dependent-test-aware regression testing techniques. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 298-311).
Detecting flaky tests in probabilistic and machine learnin	Dutta, S., Shi, A., Choudhary, R., Zhang, Z., Jain, A., & Misailovic, S. (2020, July). Detecting flaky tests in probabilistic and machine learning applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 211-224).
Know your neighbor: fast static prediction of test flakine	Verdecchia, R., Cruciani, E., Miranda, B., & Bertolino, A. (2021). Know you neighbor: Fast static prediction of test flakiness. *IEEE Access*, *9*, 76119-76134.
Modeling and Ranking Flaky Tests at Apple	Kowalczyk, E., Nair, K., Gao, Z., Silberstein, L., Long, T., & Memon, A. (2020, October). Modeling and ranking flaky tests at Apple. In 2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP) (pp. 110-119). IEEE.
Practical Automatic Lightweight Nondeterminism and Flaky Test Detection and Debugging for Python	Groce, A., & Holmes, J. (2020, December). Practical automatic lightweight nondeterminism and flaky test detection and debugging for Python. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)* (pp. 188-195). IEEE.
Shake It! Detecting Flaky Tests Caused by Concurrency with Shaker	Silva, D., Teixeira, L., & d'Amorim, M. (2020, September). Shake it! detecting flaky tests caused by concurrency with shaker. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 301-311). IEEE.
Simulating the Effect of Test Flakiness on Fault Localization Effectiveness	Vancsics, B., Gergely, T., & Beszédes, A. (2020, February). Simulating the effect of test flakiness on fault localization effectiveness. In *2020 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)* (pp. 28-35). IEEE.
Concurrency-related flaky test detection in android apps	Dong, Z., Tiwari, A., Yu, X. L., & Roychoudhury, A. (2020). Concurrency-related flaky test detection in android apps. *arXiv preprint arXiv:2005.10762*.
Empirical Study of Restarted and Flaky Builds on Travis C	Durieux, T., Le Goues, C., Hilton, M., & Abreu, R. (2020, June). Empirical study of restarted and flaky builds on Travis CI. In *Proceedings of the 17th International Conference on Mining Software Repositories* (pp. 254-264).
Flake It 'Till You Make It: Using Automated Repair to Indu	Parry, O., Kapfhammer, G. M., Hilton, M., & McMinn, P. (2020, June). Flake It'Till You Make It: Using Automated Repair to Induce and Fix Latent Test Flakiness. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops* (pp. 11-12).
Intermittently failing tests in the embedded systems dor	Strandberg, P. E., Ostrand, T. J., Weyuker, E. J., Afzal, W., & Sundmark, D. (2020, July). Intermittently failing tests in the embedded systems domain. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 337-348).
Root causing, detecting, and fixing flaky tests: state of the art and future roadmap	Zolfaghari, B., Parizi, R. M., Srivastava, G., & Hailemariam, Y. (2021). Root causing, detecting, and fixing flaky tests: state of the art and future roadmap. *Software: Practice and Experience*, *51*(5), 851-867.
Search-based crash reproduction using behavioural model seeding	Derakhshanfar, P., Devroey, X., Perrouin, G., Zaidman, A., & van Deursen, A. (2020). Search-based crash reproduction using behavioural model seeding. *Software Testing, Verification and Reliability*, *30*(3), e1733.
Understanding reproducibility and characteristics of flaky tests through test reruns in Java projects	Lam, W., Winter, S., Astorga, A., Stodden, V., & Marinov, D. (2020, October). Understanding reproducibility and characteristics of flaky tests through test reruns in Java projects. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)* (pp. 403-413). IEEE.
What is the Vocabulary of Flaky Tests?	Pinto, G., Miranda, B., Dissanayake, S., d'Amorim, M., Treude, C., & Bertolino, A. (2020, June). What is the vocabulary of flaky tests?. In *Proceedings of the 17th International Conference on Mining Software Repositories* (pp. 492-502).
A Multi-factor Approach for Flaky Test Detection and Automated Root Cause Analysis	Ahmad, A., de Oliveira Neto, F. G., Shi, Z., Sandahl, K., & Leifler, O. (2021, December). A Multi-factor Approach for Flaky Test Detection and Automated Root Cause Analysis. In *2021 28th Asia-Pacific Software Engineering Conference (APSEC)* (pp. 338-348). IEEE.
A Replication Study on the Usability of Code Vocabulary in Predicting Flaky Tests	Haben, G., Habchi, S., Papadakis, M., Cordy, M., & Le Traon, Y. (2021, May). A replication study on the usability of code vocabulary in predicting flaky tests. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)* (pp. 219-229). IEEE.
An Empirical Analysis of UI-Based Flaky Tests	Romano, A., Song, Z., Grandhi, S., Yang, W., & Wang, W. (2021, May). An empirical analysis of UI-based flaky tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (pp. 1585-1597). IEEE.
An Empirical Study of Flaky Tests in Python	Gruber, M., Lukasczyk, S., Kroiß, F., & Fraser, G. (2021, April). An empirical study of flaky tests in python. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)* (pp. 148-158). IEEE.
Empirical analysis of practitioners' perceptions of test flakiness factors	Ahmad, A., Leifler, O., & Sandahl, K. (2021). Empirical analysis of practitioners' perceptions of test flakiness factors. *Software Testing, Verification and Reliability*, *31*(8), e1791.

FlakeFlagger: Predicting Flakiness Without Rerunning Tests	Alshammari, A., Morris, C., Hilton, M., & Bell, J. (2021, May). FlakeFlagger: Predicting flakiness without rerunning tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (pp. 1572-1584). IEEE.
Flakify: A Black-Box, Language Model-based Predictor for Flaky Tests	Fatima, S., Ghaleb, T. A., & Briand, L. (2022). Flakify: A black-box, language model-based predictor for flaky tests. *IEEE Transactions on Software Engineering*.
Flaky test detection in Android via event order exploration	Dong, Z., Tiwari, A., Yu, X. L., & Roychoudhury, A. (2021, August). Flaky test detection in Android via event order exploration. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 367-378).
FLEX: fixing flaky tests in machine learning projects by up	Dutta, S., Shi, A., & Misailovic, S. (2021, August). Flex: fixing flaky tests in machine learning projects by updating assertion bounds. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 603-614).
On the Impact of Flaky Tests in Automated Program Repair	Qin, Y., Wang, S., Liu, K., Mao, X., & Bissyandé, T. F. (2021, March). On the impact of flaky tests in automated program repair. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 295-306). IEEE.
Probabilistic and systematic coverage of consecutive test-method pairs for detecting order-dependent flaky tests	Wei, A., Yi, P., Xie, T., Marinov, D., & Lam, W. (2021, March). Probabilistic and systematic coverage of consecutive test-method pairs for detecting order-dependent flaky tests. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (pp. 270-287). Springer, Cham.
Reducing Flakiness in End-to-End Test Suites: An Experience Report,	Olianas, D., Leotta, M., Ricca, F., & Villa, L. (2021, September). Reducing flakiness in End-to-End test suites: An experience report. In *International Conference on the Quality of Information and Communications Technology* (pp. 3-17). Springer, Cham.
Research Progress of Flaky Tests	Zheng, W., Liu, G., Zhang, M., Chen, X., & Zhao, W. (2021, March). Research progress of flaky tests. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 639-646). IEEE.
Discerning Legitimate Failures From False Alerts: A Study of Chromium's Continuous Integration	Haben, G., Habchi, S., Papadakis, M., Cordy, M., & Traon, Y. L. (2021). Discerning Legitimate Failures From False Alerts: A Study of Chromium's Continuous Integration. *arXiv preprint arXiv:2111.03382*.
On the use of mutation in injecting test order-dependency	Habchi, S., Cordy, M., Papadakis, M., & Traon, Y. L. (2021). On the use of mutation in injecting test order-dependency. *arXiv preprint arXiv:2104.07441*.
Tackling Flaky Tests: Understanding the Problem and Providing Practical Solutions	Gruber, M. (2021, November). Tackling Flaky Tests: Understanding the Problem and Providing Practical Solutions. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 1-3). IEEE.
TERA: optimizing stochastic regression tests in machine l	Dutta, S., Selvam, J., Jain, A., & Misailovic, S. (2021, July). Tera: Optimizing stochastic regression tests in machine learning projects. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 413-426).
Toward static test flakiness prediction: a feasibility study	Pontillo, V., Palomba, F., & Ferrucci, F. (2021, August). Toward static test flakiness prediction: A feasibility study. In *Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution* (pp. 19-24).
Verifying determinism in sequential programs (extended version)	Mudduluru, R., Waataja, J., Millstein, S., & Ernst, M. D. (2021). Verifying determinism in sequential programs (extended version). *University of Washington, Tech. Rep. TR-UW-CSE-2021-02-01*.
What is the vocabulary of flaky tests? an extended replication	Camara, B. H. P., Silva, M. A. G., Endo, A. T., & Vergilio, S. R. (2021, May). What is the vocabulary of flaky tests? an extended replication. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)* (pp. 444-454). IEEE.
What We Talk About When We Talk About Software Test Flakiness	Barboni, M., Bertolino, A., & Angelis, G. D. (2021, September). What We Talk About When We Talk About Software Test Flakiness. In *International Conference on the Quality of Information and Communications Technology* (pp. 29-39). Springer, Cham.
A Qualitative Study on the Sources, Impacts, and Mitigation Strategies of Flaky Tests	Habchi, S., Haben, G., Papadakis, M., Cordy, M., & Le Traon, Y. (2022, April). A Qualitative Study on the Sources, Impacts, and Mitigation Strategies of Flaky Tests. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)* (pp. 244-255). IEEE.
A Survey on How Test Flakiness Affects Developers and What Support They Need To Address It	Gruber, M., & Fraser, G. (2022, April). A Survey on How Test Flakiness Affects Developers and What Support They Need To Address It. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)* (pp. 82-92). IEEE.
Build System Aware Multi-language Regression Test Selection in Continuous Integration	Elsner, D., Wuersching, R., Schnappinger, M., Pretschner, A., Graber, M., Dammer, R., & Reimer, S. (2022). Build System Aware Multi-language Regression Test Selection in Continuous Integration.
Evaluating Features for Machine Learning Detection of Order- and Non-Order-Dependent Flaky Tests	Parry, O., Kapfhammer, G. M., Hilton, M., & McMinn, P. (2022, April). Evaluating Features for Machine Learning Detection of Order-and Non-Order-Dependent Flaky Tests. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)* (pp. 93-104). IEEE.
FlakiMe: Laboratory-Controlled Test Flakiness Impact Assessment	Cordy, M., Rwemalika, R., Franci, A., Papadakis, M., & Harman, M. (2022). FlakiMe: Laboratory-Controlled Test Flakiness Impact Assessment.
Flakime: Laboratory-controlled test flakiness impact assessment. A case study on mutation testing and program repair	Cordy, M., Rwemalika, R., Papadakis, M., & Harman, M. (2019). Flakime: Laboratory-controlled test flakiness impact assessment. A case study on mutation testing and program repair. *arXiv preprint arXiv:1912.03197*.
Identifying Randomness related Flaky Tests through Divergence and Execution Tracing	Ahmad, A., Held, E. N., Leifler, O., & Sandahl, K. (2022, April). Identifying Randomness related Flaky Tests through Divergence and Execution Tracing. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (pp. 293-300). IEEE.

iPFlakies: A Framework for Detecting and Fixing Python Order-Dependent Flaky Tests	Wang, R., Chen, Y., & Lam, W. (2022). iPFlakies: A Framework for Detecting and Fixing Python Order-Dependent Flaky Tests.
Preempting Flaky Tests via Non-Idempotent-Outcome Tests	Wei, A., Yi, P., Li, Z., Xie, T., Marinov, D., & Lam, W. (2022, May). Preempting Flaky Tests via Non-Idempotent-Outcome Tests. In *International Conference on Software Engineering (ICSE'22)*.
Repairing order-dependent flaky tests via test generation	Li, C., Zhu, C., Wang, W., & Shi, A. (2022). Repairing order-dependent flaky tests via test generation. ICSE.
Shaker: a Tool for Detecting More Flaky Tests Faster	Cordeiro, M., Silva, D., Teixeira, L., Miranda, B., & d'Amorim, M. (2021, November). Shaker: a Tool for Detecting More Flaky Tests Faster. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 1281-1285). IEEE.
Surveying the developer experience of flaky tests	Parry, O., Kapfhammer, G. M., Hilton, M., & McMinn, P. (2022). Surveying the developer experience of flaky tests. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*.
A Survey of Flaky Tests	Parry, O., Kapfhammer, G. M., Hilton, M., & McMinn, P. (2021). A survey of flaky tests. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *31*(1), 1-74.
Initial Results on Counting Test Orders for Order-Dependent Flaky Tests Using Alloy	Wang, W., Yi, P., Khurshid, S., & Marinov, D. (2022). Initial Results on Counting Test Orders for Order-Dependent Flaky Tests Using Alloy. In *IFIP International Conference on Testing Software and Systems* (pp. 123-130). Springer, Cham.
Static Test Flakiness Prediction	Valeria Pontillo. 2022. Static Test Flakiness Prediction. In 44th International Conference on Software Engineering Companion (ICSE '22 Companion), May 21–29, 2022, Pittsburgh, PA, USA. ACM, New York, NY, USA, 3 pages.