L'amplification de tests pour DevOps

Caroline Landry

▶ **To cite this version:**

Caroline Landry. L'amplification de tests pour DevOps. Linux Magazine France, 2019, 227, pp.1-16. hal-02418002

HAL Id: hal-02418002
https://inria.hal.science/hal-02418002

Submitted on 18 Dec 2019

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

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

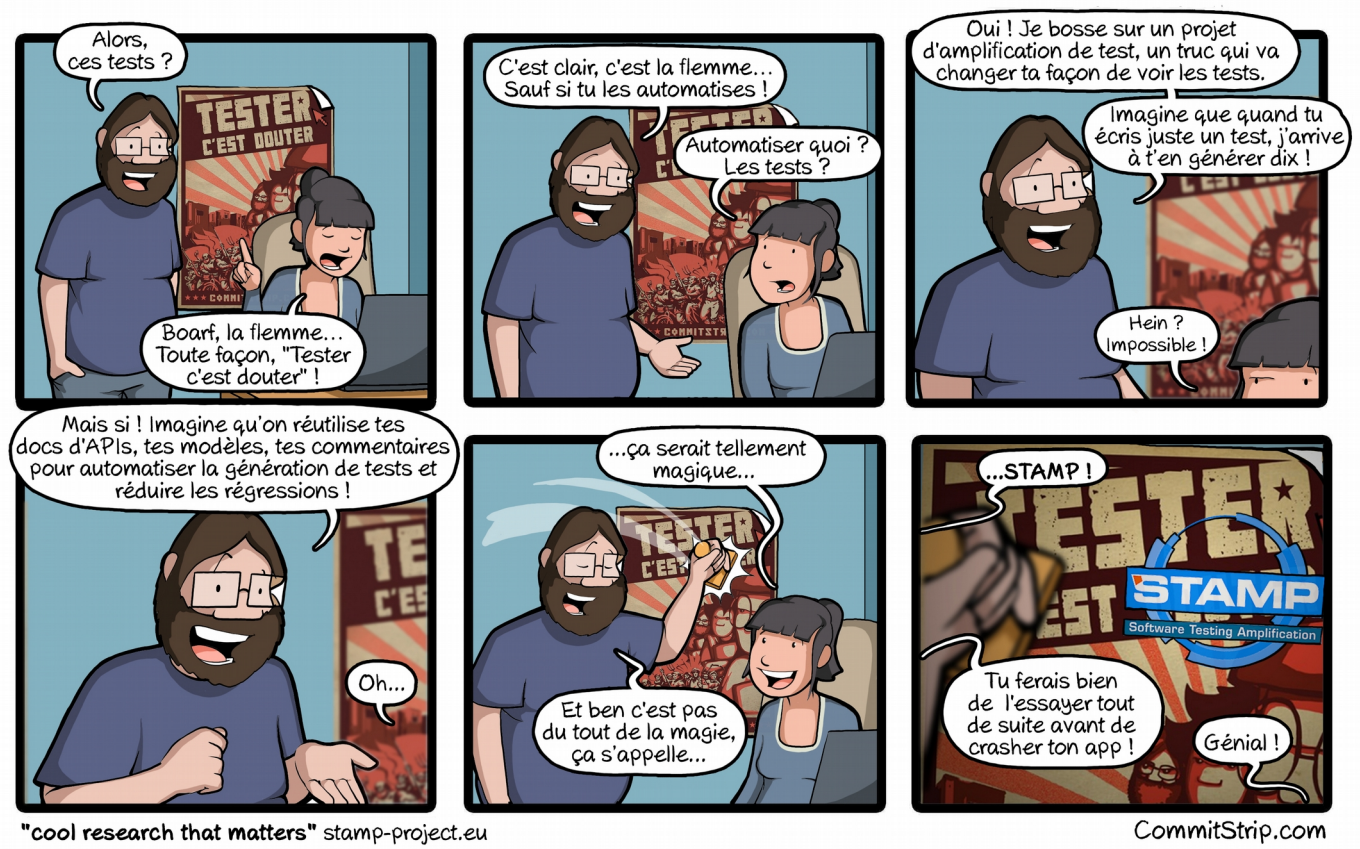
L'amplification de tests pour DevOps

Ce travail a été partiellement soutenu par le projet de l'UE STAMP ICT-16-10 No.731529 ICT-16-10 No.731529.

Caroline Landry [INRIA, France]

Mots-clés :
DevOps, test automatique, génération de test, Open Source, Java, Junit, Docker

L'amplification de tests analyse et transforme automatiquement les tests produits par les équipes de développement pour augmenter la confiance à chaque commit.



L'émergence de DevOps est une incitation très forte pour que les équipes de développement logiciel implémentent des suites de tests qui peuvent être exécutées automatiquement à chaque *commit*. Dans cet article nous discutons d'une nouvelle technique qui permet d'accompagner les développeurs dans cette démarche: l'amplification de tests. Cette approche débute avec le code produit par les développeurs : cas de tests unitaires, APIs, fichiers de configuration. Puis, l'amplification analyse et transforme automatiquement ce code pour générer des variants qui vont permettre de tester plus intensément. Dans le serveur d'intégration continue, cette technique permet d'augmenter la confiance dans le *commit*. Cet article présente quatre outils *open source* qui s'intègrent dans des chaînes de *build* et qui instancient l'idée d'amplification pour **Java** et **Junit**, et en environnement **Docker**. Ces outils sont développés par une équipe européenne dans un projet soutenu par l'UE, qui inclut des objectifs de dissémination et de transfert vers l'industrie, et une part importante de communication.

1 Test automatique et DevOps

« Publiez tôt, publiez souvent » (« Release early, release often »). Tel est le mantra des géants de l'informatique tels que **Twitter** ou **Netflix**. Pionniers dans l'ingénierie des applications web et mobiles, ils mettent en production plusieurs évolutions chaque jour. Cette agilité spectaculaire est un avantage concurrentiel majeur. Elle réduit les temps de mise sur le marché et augmente les revenus. Derrière cette prouesse se cache le concept DevOps. Cette méthodologie de développement s'appuie sur un fort taux d'automatisation à toutes les étapes de fabrication et de déploiement du logiciel.

DevOps a fait plus d'adeptes aux États-Unis qu'en Europe et on commence à craindre que les entreprises européennes ne ratent le coche. On peut penser que leur réticence reflète une différence culturelle quant à la gestion des risques. De fait, un développement hâtif peut introduire une régression en production à cause d'un manque de tests. La peur de la panne est d'autant plus justifiée que les tests dans DevOps reposent principalement sur un effort manuel.

Mettant à profit la recherche avancée en génération automatique de tests, l'**amplification de tests** a pour objectif de pousser l'automatisation de DevOps encore plus loin grâce. Cette approche utilise l'existant (cas de tests, APIs, dépendances, configurations de tests, traces d'exécution) pour générer de nouveaux tests ou de nouvelles configurations de tests à chaque mise à jour de l'application. En agissant à des étapes clefs du cycle de développement, les techniques d'amplification visent à réduire le nombre de régressions et à diminuer les coûts, au niveau des tests unitaires, des configurations de tests et de la phase de production.

2 L'amplification dans DevOps

L'amplification consiste à appliquer des transformations automatiques à un existant écrit manuellement, pour améliorer de façon significative l'efficacité des tests et détecter au plus tôt les anomalies. Cet article se focalise sur :

- les tests unitaires : détection des régressions sur les serveurs d'intégration continue avant les tests fonctionnels ;
- les tests de configuration : détecter les bugs de compatibilité (système d'exploitation, version de langage, navigateur utilisé, base de données utilisée, taille mémoire, etc) avant d'être mis en production ;
- les tests de production : reproduire les bugs de production grâce aux piles d'appels.

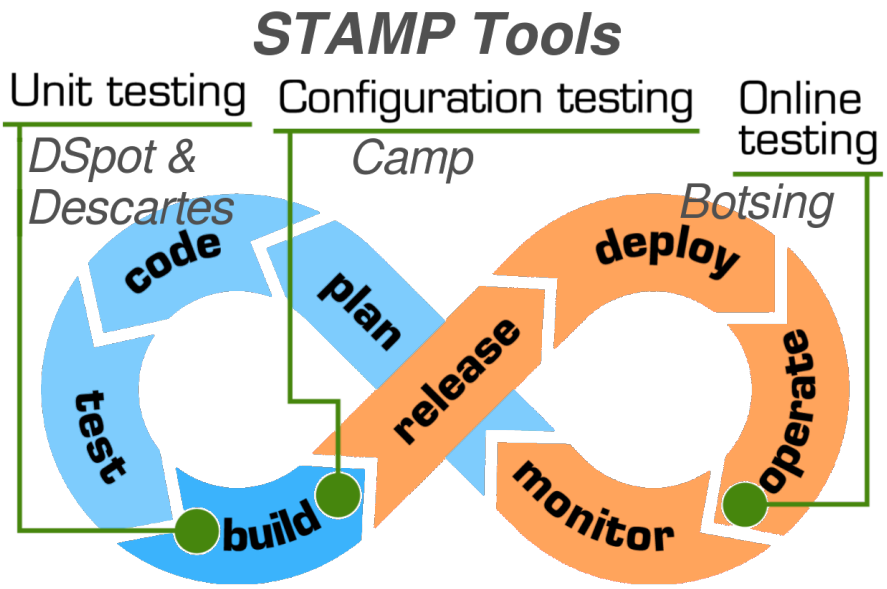


Fig. 1: Les outils d'amplification de tests dans DevOps

La figure 1 situe ces trois phases d'amplification dans DevOps. Nous présentons dans la suite, les outils qui permettent d'automatiser ces phases d'amplification. Ces techniques d'amplification sont actuellement implémentées et disponibles sous forme d'outils *open source*. Les outils **Descartes**, **Dspot** et **Botsing** travaillent sur des sources Java et des suites de tests **Junit**, cf la figure 2 montrant un exemple de test JUnit, et **CAMP** est basé sur un environnement Docker.

```java
long fact(int n) {
  if(n==0) {
    return 1;
  }
  long result = 1;
  for(int i = 2; i <= n; i++) {
   result = result * i;
  }
  return result;
}
```

```java
@Test
factorialWith5Test() {
  long obs = fact(5);
  assertTrue(5 < obs);
}


@Test
factorialWith0Test() {
  assertEqual(1, fact(0));
}
```

Fig. 2: Code source et tests JUnit associés

3 Descartes : Détecter les points faibles dans une suite de tests

3.1 Qu'est-ce que c'est ?

Descartes évalue la capacité de votre suite de tests à détecter les bogues et aide les développeurs à améliorer leurs suites de tests en signalant les points faibles du code couvert, c'est-à-dire les méthodes non testées.

Descartes est basé sur **PIT**, qui est un système de *tests par mutation* pour Java. PIT fournit un cadre pour étendre ses fonctionnalités de base en utilisant des *plugins*, et Descartes est donc un *plugin* de PIT qui implémente un moteur de *tests par mutation extrême*.

3.2 Comment ça marche ?

3.2.1 Tests par mutation

Les tests par mutation vous permettent de vérifier si vos suites de tests peuvent détecter d'éventuels bugs, en introduisant de petits changements ou des fautes dans le programme original. Ces versions modifiées sont appelées mutants (voir figure 3). Une bonne suite de tests doit être capable de tuer ou de détecter les mutants.

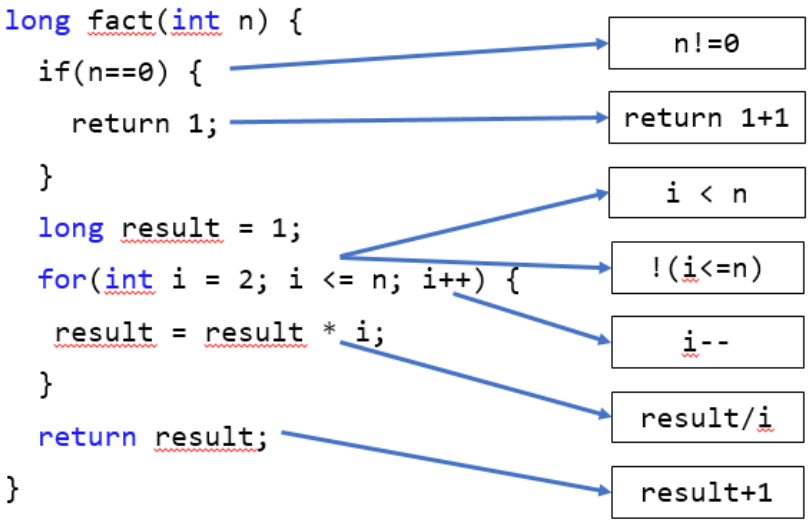


Fig. 3: Exemple de fonction et de ses mutants traditionnels

Le test par mutation traditionnel fonctionne au niveau de l'instruction, par exemple en remplaçant **>** par **≤**, de sorte que le nombre de mutants générés est énorme, tout comme le temps nécessaire pour vérifier l'ensemble de la suite de tests. C'est une des raisons pour lesquelles les auteurs de « Will my tests tell me if I break this code ? » **[1]** ont proposé une stratégie de tests par mutation extrême, qui

fonctionne au niveau de la méthode. Une autre raison était de résoudre le problème des mutants équivalents. Un mutant équivalent est un mutant qui est équivalent au programme original et n'est donc pas différenciable du programme original. La détection automatique des mutants équivalents est en général un problème indécidable.

3.2.2 Tests par mutation extrême

Lors de l'analyse de mutation extrême, toute la logique d'une méthode testée est éliminée. Pour les procédures, toutes les instructions sont simplement supprimées, et pour les fonctions toutes les instructions sont remplacés par une valeur de retour constante (voir figure 4).

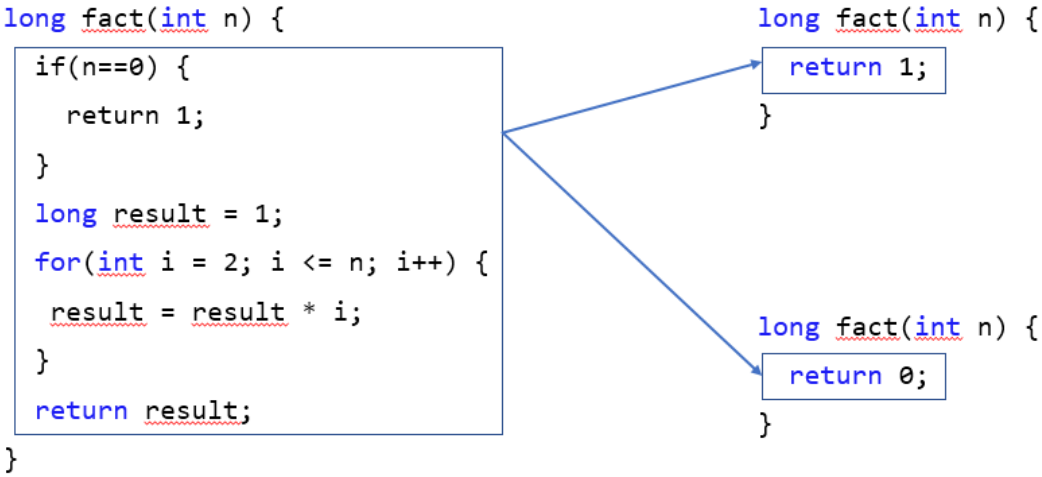


Fig. 4: Exemple de fonction et de ses mutants extrêmes

Cette approche présente plusieurs avantages :

- elle crée beaucoup moins de mutants et donc permet une analyse beaucoup plus rapide ;
- elle travaille au niveau méthode et est ainsi plus facile à comprendre ;
- elle permet de détecter la plupart des mutants équivalents, ce qui la rend plus fiable.

3.3 Résultats produits

Descartes est un *plugin* PIT. Ainsi, les résultats fournis sont les mêmes que PIT, à savoir :

- un score de couverture de code : la proportion du code testé ;
- un score de mutation : la proportion de mutants tués ;
- un rapport détaillé (voir figure 5) sur la couverture des lignes et la couverture des mutations pour chaque fichier source.

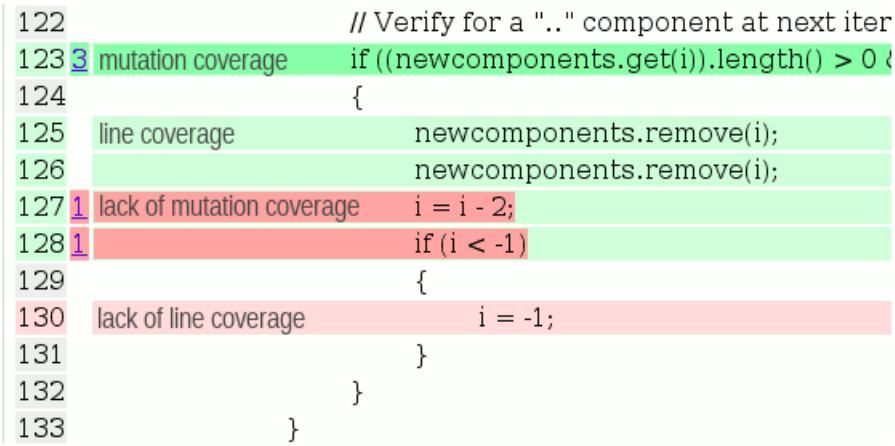


Fig. 5: Rapport détaillé de PIT

Mais Descartes identifie en plus :

- les méthodes pseudo-testées (voir figure 6) : méthodes exécutées par la suite de tests et pour lesquelles aucune mutation extrême n'est détectée ;

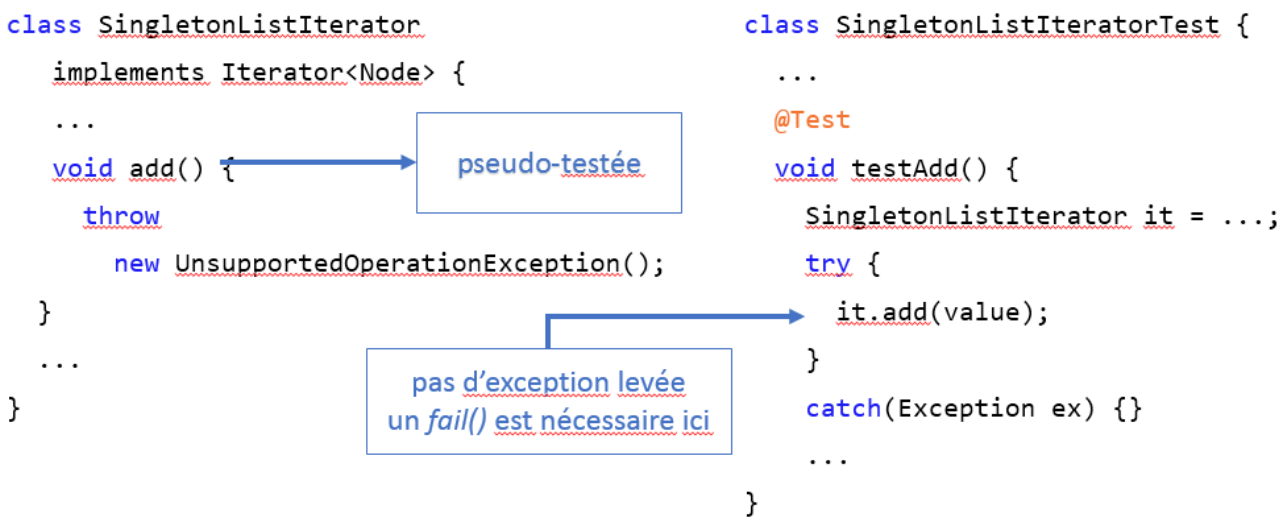


Fig. 6: Méthode pseudo-testée

- les méthodes partiellement testées (voir figure 7) : méthodes exécutées par la suite de test et pour lesquelles toutes les mutations extrêmes ne sont pas détectées.

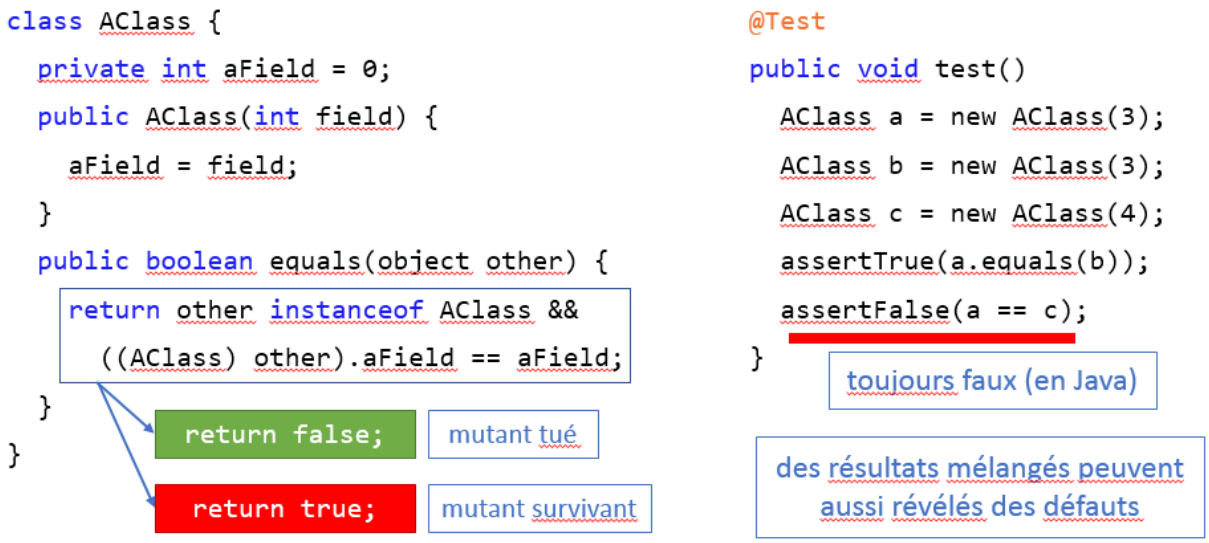


Fig. 7: Méthode partiellement testée

Des expérimentations ont été menées par l'**INRIA** et **KTH [2] [3]** sur le test par mutation extrême sur un ensemble de projets *open source* (voir la figure 8 présentant un extrait des résultats des expérimentations de Descartes). Ces expériences ont démontré un intérêt de la mutation extrême pour les équipes de développement. Elles ont également montré que la mutation extrême est aussi efficace que la mutation traditionnelle pour détecter les points faibles dans une suite de tests.

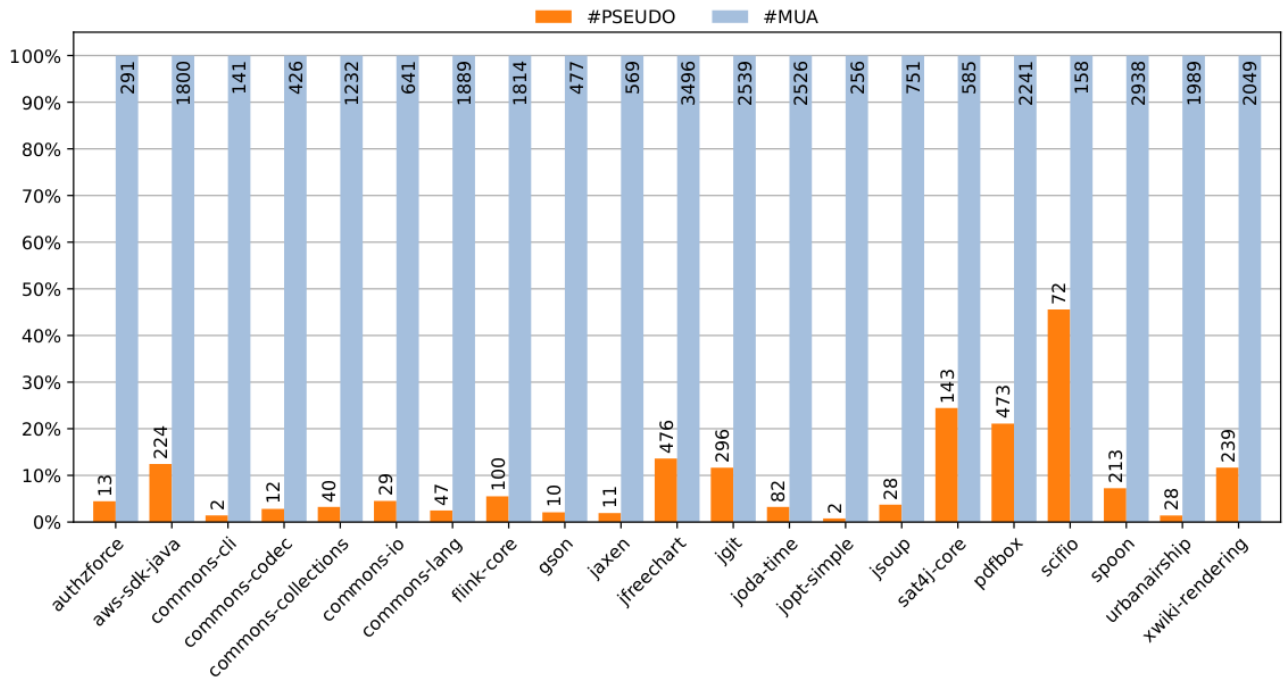


Fig. 8: Nombres de méthodes testées vs pseudo-testées sur des projets open source

Les principaux points à retenir sur le test par mutation extrême sont :

- génère moins de mutants et l'analyse est plus rapide ;
- ne remplace pas le test par mutation traditionnel ;
- les méthodes pseudo-testées peuvent révélées des bugs dans les tests ;
- d'après les développeurs 1/3 des méthodes pseudo-testées sont pertinentes.

3.4 Utiliser Descartes

Descartes est intégré à **Maven** (disponible dans **Maven Central**), **Gradle** et **Eclipse**, et disponible en ligne de commande.

C'est un outil *open source* disponible dans le dépôt **GitHub** du projet **STAMP** : **https://github.com/STAMP-project/pitest-descartes**.

4 DSpot : amplification de cas de tests unitaires

4.1 Qu'est-ce que c'est ?

DSpot améliore automatiquement les suites de test JUnit existantes. Il génère automatiquement soit de nouvelles assertions dans les cas de tests existants, soit de nouveaux cas de tests.

4.2 Comment ça marche ?

Le critère d'évaluation sur lequel s'appuie DSpot est le score de mutation. Il permet de déterminer si un variant de cas de test est utile : il ne sélectionne que des cas de tests qui couvrent des branches non couvertes par les tests originaux ou qui vérifient des états non vérifiés par les tests originaux. Le choix du score de mutation par rapport au taux de couverture permet de valider que les nouvelles assertions sont utiles pour observer des états qui ne sont pas observés par les tests originaux.

DSpot instrumente le code des tests et génère des variants de tests à partir des tests existants pour créer de nouveaux tests ou de nouvelles assertions dans les tests existants. Il sélectionne ensuite les tests à garder en utilisant le critère d'évaluation pour générer une suite de tests amplifiée, comme illustré dans la figure 9.

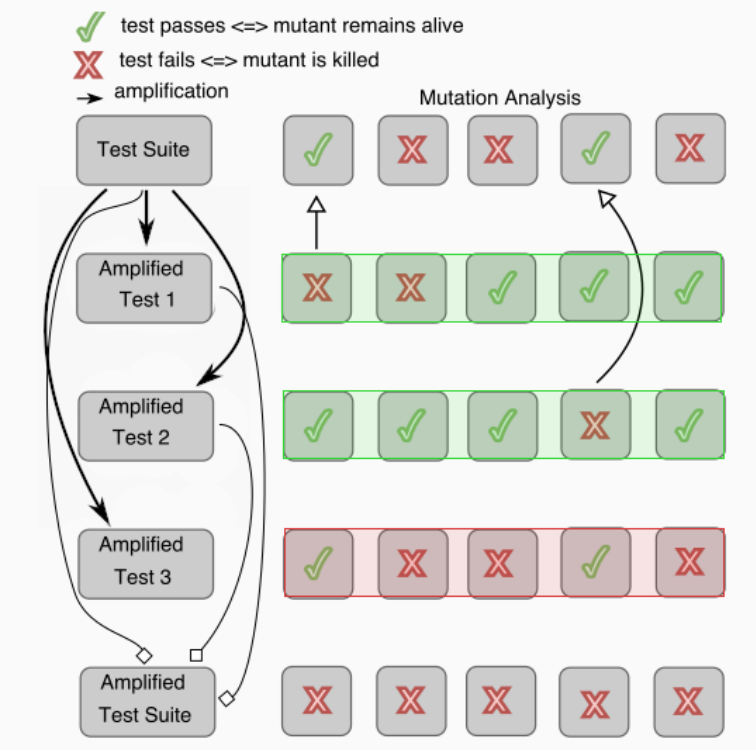


Fig. 9: Sélection des tests amplifiés et génération de la suite de tests amplifiée.

La suite amplifiée tue plus de mutants et donc le score de mutation a augmenté.

4.3 Résultats produits

DSpot prend en entrée un programme Java et sa suite de tests, et génère une suite de tests amplifiée qui améliore le critère qui permet d'évaluer la suite de test (voir figure 10). La figure 11 montre un exemple de test amplifié.

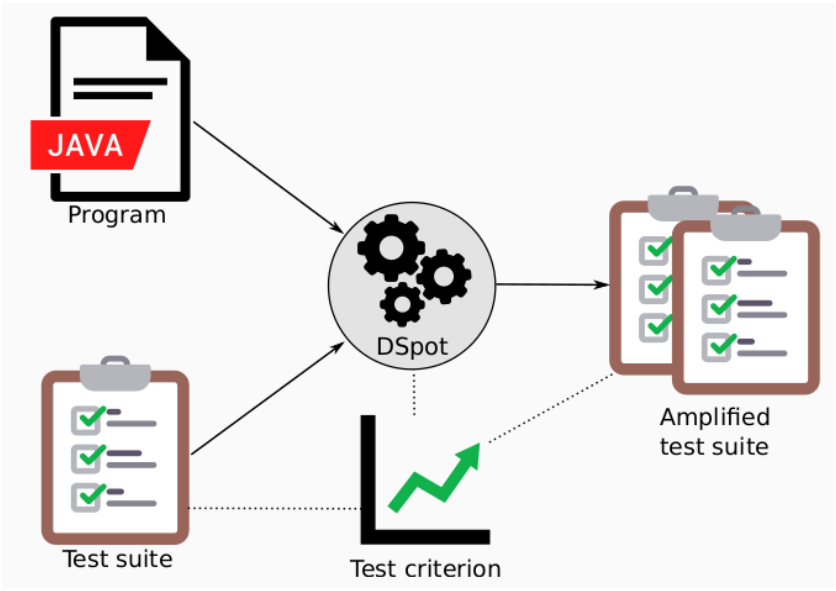


Fig. 10: Entrés et sortie de Dspot

Original test case

```
@Test
public void html() {
  Food tacos = new Food("Tacos");
  Student benjamin = new Student("Benjamin");
  benjamin.eat(tacos);

  assertFalse(benjamin.isHungry());
}
```

```
@Test
public void html() {
  Food tacos = new Food("Tacos");
  Student benjamin = new Student("Benjamin");

  assertTrue(benjamin.isHungry());
  assertFalse(benjamin.isHappy());
}
```

Amplified test case

Fig. 11: Exemple de test amplifié

Des expérimentations ont été menées par l'INRIA et KTH **[6] [3]** en appliquant DSpot sur un ensemble de projets *open source*, et les résultats obtenus ont permis de générer des « Pull Requests » pour améliorer les tests de ces projets. La figure 12 résume l'essentiel de cette expérience : DSpot a pu amplifier les tests existants de manière à fournir des ajouts jugés significatifs par les développeurs (qui ont accepté la *pull request*).

project	# opened	# merged	# closed	# under discussion
javapoet	4	4	0	0
mybatis-3	2	2	0	0
traccar	2	1	0	1
stream-lib	1	1	0	0
mustache	2	2	0	0
twilio	2	1	0	1
jsoup	2	0	1	1
prostostuff	2	2	0	0
logback	2	0	0	2
retrofit	0	0	0	0
total	19	13	1	5

Fig. 12: État des Pull Requests contruites à partir des résultats de DSpot.

4.4 Utiliser DSpot

DSpot est intégré à Maven (disponible dans Maven Central), Gradle et Eclipse, et disponible en ligne de commande.

C'est un outil *open source* disponible dans le dépôt GitHub du projet STAMP : **https://github.com/STAMP-project/dspot**.

5 CAMP : amplifier les configurations pour le test d'intégration

5.1 Qu'est-ce que c'est ?

Nos programmes s'exécutent dans un environnement, que ce soit directement sur le processeur, sur le système d'exploitation, sur des plateformes telles que Java, **Python** ou **Ruby** ou encore sur des serveurs d'applications tels que **Tomcat** par exemple. Plus cet environnement est complexe, plus il est difficile d'anticiper son influence sur notre code. Par ailleurs, cet environnement évolue : les versions actuelles ne seront pas toujours supportées, de plus performantes vont apparaître ou d'autres produits vont s'imposer.

Les tests qui prennent en compte cet environnement sont dits « d'intégration » ou « système » : ils s'opposent aux tests dits « unitaires » qui vérifient de plus petits morceaux de code (fonctions, méthodes, classes, etc.) en isolation (pas de base de données, de système de fichiers, de connexions réseau, etc.). Ces tests d'intégration nécessitent de recréer artificiellement la configuration, ou autrement dit l'environnement, avant d'y déployer notre application et de finalement exécuter nos tests.

Avec CAMP, notre objectif est d'amplifier ces tests d'intégration. À partir d'une configuration de base et d'alternatives possibles, CAMP exécute nos tests dans toutes les configurations possibles.

5.2 Comment ça marche ?

Pour faire varier notre environnement, CAMP a besoin d'un modèle de déploiement (« template » en anglais) ainsi que d'une description des éléments à y modifier. Ce modèle est une orchestration de services qui inclut tous les fichiers nécessaires pour construire et déployer notre application.

Il s'accompagne d'un fichier **YAML** décrivant les différents éléments à y modifier. On peut voir dans l'exemple ci-après que la fonctionnalité `ruby` a 1 instanciation possible avec 2 variables `version` et `max_alloc` avec la spécification de leur domaine de définition (`values`), et que le service `DB` a quant à lui 2 instanciations possibles.

```
goals:
  running:
    - Registry

components:

  registry:
    provides_services: [ Registry ]
    requires_services: [ DB ]
    requires_features: [ Ruby ]
    implementation:
      docker:
        file: registry/Dockerfile

  ruby:
    provides_features: [ Ruby ]
    variables:
      version:
        type: Text
        values: [ v2.4, v2.5 ]
        realization:
          - targets: [ ruby/Dockerfile ]
            pattern: "FROM ruby:2.5.1-alpine3.7"
            replacements: [ "FROM ruby:2.4.1-alpine3.7", "FROM ruby:2.4.5-alpine3.7" ]
      max_alloc:
        type: Integer
        values: [ 8, 16, 32, 64 ]
        realization:
          - targets: [ ruby/config.rb ]
            pattern: ":malloc_limit=>16777216"
            replacements:
             - ":malloc_limit=>8388608"
             - ":malloc_limit=>16777216"
             - ":malloc_limit=>33554432"
             - ":malloc_limit=>67108864"
    implementation:
      docker:
        file: ruby/Dockerfile

  mysql:
    provides_services: [ DB ]
    variables:
```

```
    version:
      type: Text
      values: [ v5.6, v5.7]
      realization:
        - targets: [ docker-compose.yml ]
          pattern: "image:mysql:5.7"
          replacements: [ "image:mysql:5.6", "image:mysql:5.7" ]
  implementation:
    docker:
      image: mysql:5.6

mariadb:
  provides_services: [ DB ]
  variables:
    version:
      type: Text
      values: [ v5.6, v10.2 ]
      realization:
        - targets: [ docker-compose.yml ]
          pattern: "image:mysql:5.7"
          replacements: [ "image:mariadb:5.6", "image:mariadb:10.2" ]
  implementation:
    docker:
      image: mariadb:5.6
```

CAMP modifie l'environnement de trois manières différentes : il modifie l'orchestration des services (1), l'agencement des piles logicielles sous-jacentes (2), ainsi que la configuration détaillée des *components* (3).

1. L'orchestration de services définit quels éléments seront déployées dans des conteneurs séparés, et potentiellement distribués. La figure 13 présente un exemple d'orchestration où une application web nommée « Registry » utilise une base de données, soit **MySQL** soit **MariaDB**.

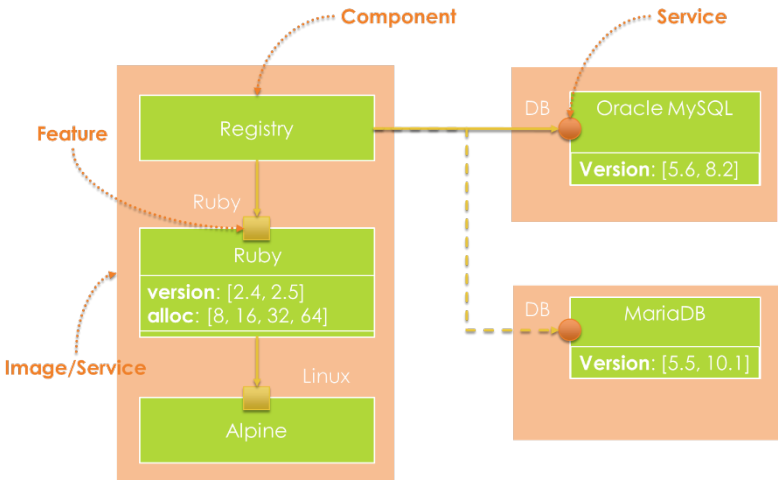


Fig. 13: Exemple : application « Registry » en Ruby connectée à une base de données MySQL

Une même image contient l'application « Registry » et ses dépendances directes. CAMP construit donc des orchestrations avec deux services, le premier incluant l'application Ruby et ses dépendances, le second la base de données sélectionnée, soit MySQL, soit MariaDB. L'orchestration complète est par exemple définie dans ce *Docker Compose File* :

```
version: '3'

services:

  tests:
    image: "fchauvel/sensapp-tester:v0.1.1"
    command: echo "ready"
    volumes:
      - ./test-results:/var/results
    depends_on:
      - receiver
      - registry

  receiver:
    image: "fchauvel/sensapp-receiver:v0.1.0"
    command: node app/start -q task-queue -p 5672 -n SENSAPP_QUEUE
    expose:
```

```
      - "3000"
    depends_on:
      - task-queue

  task-queue:
    image: "rabbitmq:3.7.7"
    expose:
      - "5672"

  storage:
    image: "fchauvel/sensapp-storage:v0.1.0"
    tty: true
    command: sensapp-storage -q task-queue -p 5672 -n SENSAPP_QUEUE -o storage-db -r 8086
    depends_on:
      - task-queue
      - storage-db
      - registry

  storage-db:
    image: "influxdb:1.6.1"
    expose:
      - "8086"

  registry:
    image: fchauvel/sensapp-registry:v0.1.0
    command: ruby app/app.rb -h registry-db
    expose:
      - "4567"
    depends_on:
      - registry-db

  registry-db:
    image: mysql:5.7
    command: --default-authentication-plugin=mysql_native_password
    expose:
      - "3306"
    environment:
      MYSQL_ROOT_PASSWORD: 123456
```

Complété par le **Dockerfile** de l'application « Registry » :

```
FROM ruby:2.5.1-alpine3.7

LABEL maintainer "franck.chauvel@sintef.no"

WORKDIR /registry
COPY . /registry

# Install the tools and
RUN apk add --no-cache build-base mariadb-dev \
    && gem install bundler \
    && bundler install --without test . \
    && apk del build-base

# Run sensapp-storage
CMD ["ruby", "app/app.rb"]
```

2. Notre application « Registry » s'exécute sur l'interpréteur Ruby, qui s'exécute lui-même sur le système d'exploitation. Ces trois composants forment une pile logicielle que CAMP pourra faire varier, en modifiant par exemple la version de l'interpréteur Ruby.

3. Finalement, CAMP peut aussi modifier certains paramètres de configuration, tels que la quantité maximale de mémoire que l'interpréteur Ruby peut allouer d'un bloc (quatre valeurs sont autorisées: **8**, **16**, **32** et **64** MB). Ces paramètres influent principalement sur les performances mais sont aussi parfois la source de problèmes inattendus.

CAMP utilise **Z3**, un moteur de résolution de contraintes pour trouver les configurations possibles, vu qu'il peut exister des contraintes qui invalident certaines combinaisons. Pour chaque configuration, CAMP copie le modèle d'orchestration et y substitue des valeurs associées à chaque paramètre sélectionné par Z3.

5.3 Résultats produits

CAMP génère des environnements prêts à être déployés directement à l'aide de Docker. Dans le cas de l'application « Registry » que nous avons présentée, CAMP génère les 32 configurations possibles. Il peut aussi en générer un sous-ensemble qui assure seulement que toutes les variations sont représentées au moins une fois, comme montré dans la figure 14.

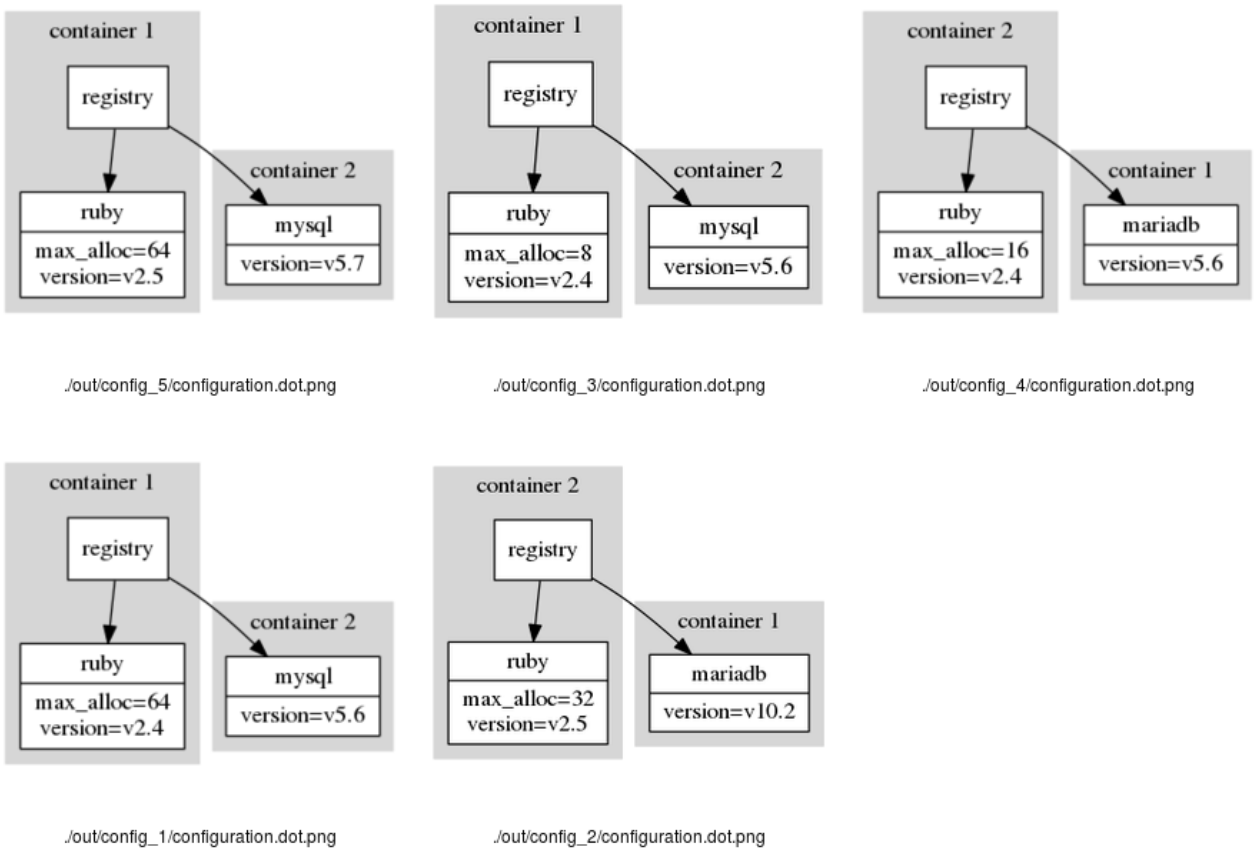


Fig. 14: CAMP génère cinq configurations pour couvrir les variations de l'application « Registery ».

CAMP nous permet ainsi de rapidement vérifier la portabilité de nos applications en amplifiant un test d'intégration. Associé à la plateforme Docker, il devient possible de tester rapidement de nombreux environnements différents, que ce soit pour tester de nouvelles versions ou de nouveaux choix architecturaux. CAMP permet donc de détecter des bugs spécifiques à certaines configurations, cf figure 15.

*Fig. 15: Bugs XWiki spécifiques à certaines configurations **[5]**.*

Comme ces environnements sont aussi décisifs pour les performances de nos applications, les travaux futurs s'orientent vers l'analyse des performances et la recherche automatique des configurations optimales d'exécution d'une application.

5.4 Utiliser CAMP

CAMP un outil *open source* disponible dans le dépôt GitHub du projet STAMP : **https://github.com/STAMP-project/camp**.

6 Botsing (TUDelft)

6.1 Qu'est-ce que c'est ?

Botsing (qui signifie crash en néerlandais) génère automatiquement un test capable de reproduire un crash, sur base de la *stack trace* (trace d'appels) Java produite par le programme lors du crash. Par exemple, la figure 16 présente une trace d'appels rapportée par les utilisateurs de **XWiki** via **JIRA**. Le test produit aide au *debug* de l'application et sert de base à la rédaction d'un test de non-régression.

```
java.lang.IllegalArgumentException: An Entity Reference name cannot be null or empty
    at org.xwiki.model.reference.EntityReference.setName(EntityReference.java:186)
    at org.xwiki.model.reference.EntityReference.<init>(EntityReference.java:154)
    at org.xwiki.model.reference.AttachmentReference.<init>(AttachmentReference.java:68)
    at com.xpn.xwiki.doc.XWikiAttachment.getReference(XWikiAttachment.java:156)
    [...]
```

Fig. 16: Extrait d'une trace d'appels rapportée par les utilisateurs de XWiki (Issue #13196).

6.2 Comment ça marche ?

Botsing utilise un algorithme d'optimisation, appelé algorithme génétique, pour générer et raffiner itérativement un ensemble de tests unitaires, jusqu'à la production d'un test capable de reproduire la trace d'appel Java donnée. Les algorithmes génétiques sont des méta-heuristiques inspirées de la théorie de l'évolution. L'idée est simple : seuls les plus adaptés survivent. Lors de l'exécution d'un algorithme génétique, une population d'individus (ici, des tests unitaires) évolue en subissant un processus de sélection naturelle. À chaque génération, les individus les plus adaptés sont sélectionnés pour faire partie de la génération suivante, en subissant éventuellement une mutation génétique ou un croisement (*crossover*) entre deux individus. L'ensemble du processus est dirigé par une fonction d'évaluation, donnant pour un individu son degré d'adaptation.

Dans le cas de Botsing, la fonction d'évaluation se base sur les lignes couvertes par un test et la (non) génération d'une exception et de sa trace d'appels. Plus un test se rapproche de la ligne où le crash s'est produit dans le code source, plus le test sera jugé adapté. De manière similaire, si le test est capable de générer une exception avec une trace d'appels similaire à celle donnée, il sera jugé plus adapté qu'un test ne générant aucune exception.

6.3 Résultats produits

Le résultat de Botsing est un test unitaire reproduisant un crash. Ce test peut alors être utilisé, par exemple en mode *debug*, afin de comprendre ce qu'il se passe dans l'application. Par exemple, pour la trace de la figure 16, Botsing produit le test de la figure 17 qui déclenche une exception et reproduit la même trace.

```
@Test(timeout = 4000)
public void test0()  throws Throwable  {
    XWikiDocument xWikiDocument0 = mock(XWikiDocument.class, new ViolatedAssumptionAnswer());
    doReturn((DocumentReference) null).when(xWikiDocument0).getDocumentReference();
    XWikiAttachment xWikiAttachment0 = new XWikiAttachment(xWikiDocument0, "");
    xWikiAttachment0.getReference();
}
```

Fig. 17: Test unitaire Java généré par Botsing pour la trace d'appel de la Figure 17

Botsing a fait l'objet de différentes expériences à l'**Université Technique de Delft**, aux Pays-Bas **[4]** lors desquelles Botsing a été utilisé pour reproduire un ensemble de crashs issus de projets *open source*. La figure 18 présente le nombre de crash reproduits automatiquement dans certains projets étudiés. Ces crashs ont été extraits des logiciels de suivi de problèmes (*issue trackers*) associés à ces projets et compilés dans un *benchmark* accessible à tous :

https://github.com/STAMP-project/JCrashPack. La dernière version incluant les développements les plus récents est disponible dans le dépôt du projet.

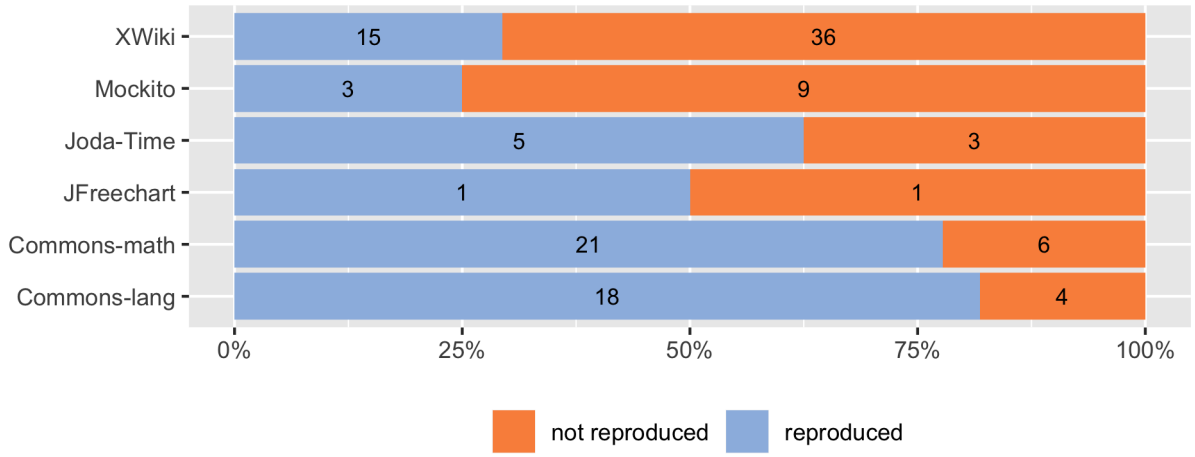


Fig. 18: Nombres de crashs issus de projets open source reproduits

Ce qu'il faut retenir à propos de Botsing :

- Botsing génère automatiquement un test reproduisant un crash sur base de sa trace d'appel ;
- pour se faire, Botsing utilise un algorithme génétique qui explore le code source de l'application et fait évoluer un ensemble de tests pour arriver à un test répliquant le crash ;
- le test produit aide le développeur à identifier la source du crash pour corriger l'application et écrire un test de non-régression.

6.4 Utiliser Botsing

Botsing est intégré à Maven (disponible dans Maven Central) et Gradle, et disponible en ligne de commande.

C'est un outil *open source* disponible dans le dépôt GitHub du projet STAMP : **https://github.com/STAMP-project/botsing**.

7 Le consortium STAMP

Le projet rassemble 4 partenaires académiques avec une forte expérience en test logiciel, 5 entreprises de différents secteurs (e-santé, gestion de contenu, ville intelligente et administration public) et 1 consortium *open source* (voir **https://www.stamp-project.eu/view/main/consortium**). Le rôle des partenaires académiques est de développer les outils et d'en mesurer l'adéquation aux besoins en répondant aux questions :

- Quelles sont les amplifications efficaces ?
- Dans quelle mesure l'amplification améliore la qualité des tests ?
- Les résultats amplifiés sont-ils utiles aux développeurs ?
- Comment l'amplification s'intègre-t-elle dans le cycle de vie ?

Le rôle des autres partenaires est double :

- fournir les cas d'utilisation pour tester les outils et effectuer les mesures d'adéquation aux besoins ;
- industrialiser les outils et les intégrer dans leurs chaînes de fabrication.

Les cas d'utilisation

Les partenaires industriels ont fourni des cas d'utilisation réels dans des domaines d'applications variés :

- **Activeon** : **Proactive Workflows and Scheduling** est une suite logicielle qui permet aux utilisateurs de créer et d'exécuter des ensembles de tâches avec des dépendances ;
- **ATOS** : **CityGo** est un ensemble de services logiciels dans le domaine des villes intelligentes, développés dans le contexte de l'écosystème **FIWARE** (**https://www.fiware.org/**).
- **TellU** : **TelluCloud** fournit des services IoT dans les domaines de la technologie du bien-être et de la santé en ligne.
- **XWiki**: XWiki est un système de gestion de contenu *open source* avec des développements collaboratifs.
- **OW2** : OW2 est un consortium *open source* et permet d'avoir accès à différents projets, notamment **Authzforce** (Contrôle d'accès) et **SAT4J** (Moteur de résolution par contraintes).

Leur objectif étant d'intégrer des outils STAMP dans leurs différents processus opérationnels :

- serveur d'intégration continue et de déploiement continue,
- traitement automatique des crashs chez les clients,
- intégrations dans leur programme qualité et dans les métriques de projet.

Conclusion

Plusieurs technologies qui permettent de réduire les temps de développement des tests sont déjà matures. Les outils STAMP sont *open source* (**https://github.com/STAMP-project**), intégrables dans les outils d'intégration continue et avec les chaînes de *build* les plus utilisées.

Et bien entendu, les développements et les recherches continuent pour améliorer les outils :

- Descartes : proposition de nouveaux tests ou d'amélioration des tests existants pour chaque méthode pseudo-testée ou partiellement testée ;
- DSpot : amélioration des performances et de la lisibilité des tests générés ;
- CAMP : tests de performance et détermination des configurations optimales ;
- Botsing : amélioration des capacités de génération pour prendre en compte d'avantages de types de crashs, notamment en se basant sur d'autres informations que celles présentes dans la trace d'appel (par exemple, des logs applicatifs).

Une campagne de bêta tests (*https://www.stamp-project.eu/view/main/betatesting/*) est en cours, alors n'hésitez pas à nous rejoindre pour nous donner votre avis sur les outils STAMP ! :-)

Remerciements

Un grand merci à mes collègues du projet STAMP :

- Benjamin Danglot et Oscar Luis Vera Pérez [INRIA, France] ;
- Benoit Baudry et Martin Monperrus [KTH, Suède] ;
- Franck Chauvel et Brice Morin [SINTEF, Norvège] ;
- Pouria Derakhshanfar, Xavier Devroey, Arie van Deursen, et Andy Zaidman [TU Delft, Pays-Bas].

Pour leur contribution, et plus généralement pour notre collaboration, c'est vraiment un plaisir de travailler avec eux.

Références

[1] R. NIEDERMAYR, E. JUERGENS, S. WAGNER, *« Will my tests tell me if I break this code? »*, Proceedings of the International Workshop on Continuous Software Evolution and delivery, ACM Press, New York, pp 23–29, 2016.

[2] O. VERA-PÉREZ, B. DANGLOT, M. MONPERRUS, B. BAUDRY, *« A comprehensive study of pseudo-tested methods »*, Empirical Software Engineering, 2018 : **https://doi.org/10.1007/s10664-018-9653-2**.

[3] O. VERA-PÉREZ, M. MONPERRUS, B. BAUDRY, *« Descartes: a PITest engine to detect pseudo-tested methods: tool demonstration »*, Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018 : **http://doi.org/10.1145/3238147.3240474P**.

[4] B. DANGLOT, O. VERA-PÉREZ , B. BAUDRY, et M. MONPERRUS, *« Automatic Test Improvement with Dspot: a Study with Ten Mature Open-Source Projects »*.

[5] M. SOLTANI, P. DERAKHSHANFAR, A. PANICHELLA, X DEVROEY, A. ZAIDMAN et A. VAN DEURSEN, *« Single-objective Versus Multi-objectivized Optimization for Evolutionary Crash Reproduction »*, Symposium on Search-Based Software Engineering - SSBSE (Montpellier, France), 325–340, 2018.

[6] **https://jira.xwiki.org/browse/XWIKI-15846?jql=labels%20%3D%20configurationtesting%20**

Pour aller plus loin

Le GitHub du projet STAMP : **https://github.com/STAMP-project**.

Le site du projet : **https://www.stamp-project.eu**.