Université de Namur
Faculté d'informatique
Année académique 2021–2022

Génération de tests unitaires pour programmes python

Ortegat Pierre

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Xavier Devroey

Co-promoteur : Benoît Vanderose

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

Remerciements

Résumé

Table des matières

Chapitre 1

Introduction

Chapitre 2

État de l'art

2.1 Méthodes de test

Cet état de l'art couvre une série de méthode de test de composant logiciels, qu'il soient auto-contenus ou distribués. Tous ceux listés dans cet état de l'art ont été considérés pour être intégré dans le travail décris après. Ils n'ont su être tous intégrés pour des raisons décrites plus loin.

2.1.1 Tests manuels

Pour des raisons de complétude, il est intéressant d'inclure dans cette liste de méthodes et techniques de test la façon la plus ancienne qui existe : l'écriture et l'utilisation de tests unitaires manuels. Ces tests, bien que plus ciblé et direct que les automatiques ne sont pas sans défauts.

Ils sont plus lents à écrire et nécessitent beaucoup plus de ressources que leurs contreparties automatiques. De par la nature humaine de leur création, ils ne seront pas toujours adaptés, bien conçus ou n'auront pas une bonne couverture du code et des fonctionnalités mais pourront être plus concis dans le cas de test précis [60].

Un autre problème récurrent rencontré lors de l'utilisation de tests unitaires écrits à la main est un problème de biais de confirmation. En effet, il n'est pas rare d'observer que les tests unitaires écrits soient juste une confirmation de ce que le développeur a écrit et non ce qui a été demandé comme fonctionnalité [19]. Ce biais de confirmation rend l'utilisation de ces tests unitaire inutile car la seule chose qu'ils vont confirmer est que l'implémentation du développeur initial est bien celle utilisée. Une méthode utilisée pour combattre ce phénomène est de faire du développement par le comportement (appelée BDD = Behavior Drive Developpement dans l'industrie) qui va consister à écrire les tests avant d'écrire le code et, idéalement, que ces tests initiaux soient écrits par quelqu'un d'extérieur au développeur de la fonctionnalité.

Un autre problème classique récurent des tests unitaires écrits à la main est la "flackiness" ($\approx$ non fiabilité et instabilité). Ce problème diminue fortement l'efficacité des-dits tests, car ils ne permettent plus de s'appuyer dessus afin de garantir que le software fonctionne correctement. Des erreurs à la fréquence assimilable à de l'aléatoire, n'apportent aucune information pertinente supplémentaire [38].

Ces divers problèmes et défauts des tests unitaires écrits à la main ont mené au développement d'autres techniques dont le fuzzing qui va être décrit dans les sections suivantes.

2.1.2 Fuzzing

Lors de son invention, le terme "Fuzzing" avait pour signification de fournir des entrées aléatoires à un programme, dans le but d'en augmenter la robustesse [24].

Aujourd'hui, les pratiques regroupées sous le terme fuzzing sont nombreuses et variées et servent plusieurs buts différents. L'utilisation la plus classique de cette technique est la sécurisation des parties de logicielle qui interagissent avec des entrées extérieures [27] (on peut par exemple citer le fait que du fuzzing est une des étapes incluses dans le "Microsoft Security Development Lifecycle" [34]). Avoir fuzzé une interface logicielle ne garantit pas que cette interface est sans faille ni bugs mais permet de raisonnablement considérer que l'interface est sécurisée et sans bugs[27].

Fuzzing en boite noire

Le fuzzing dit "en boite noire" est une des approches existante du fuzzing qui va considérer trois éléments lors de chaque test :

- Les entrées fournies au programme
- Les réponses données par le programme (*toutes* les sorties et effets de bord du programme sont pris en compte ; cela inclus les erreurs)
- Le programme en lui-même qui va être considéré comme une unité atomique, aucune observation ne sera faite sur l'état interne du programme.

Des entrées vont être générées à l'aveugle, fournies au programme et le résultat va être enregistré [24]. Si le résultat n'est pas satisfaisant, les entrées qui ont déclenché ce résultat vont être remontées pour qu'une correction puisse être effectuée.

Le principal défaut de cette méthode de test est qu'elle est fortement dépendante de la qualité des entrées initialement fournies ou générées.

Par exemple : le cas d'un programme très simple qui ne prend qu'un paquet Ethernet en entrée. Le contenu maximum d'une trame Ethernet est de 1500 bytes (pour l'exemple, on considère que la trame Ethernet est elle-même bien formée). Pour couvrir l'intégralité des probabilités d'entrée du programme, il est nécessaire de générer et de tester le nombre suivant de possibilités :

$$2^{1500*8} \approx 2,2905 * 10^{3612}$$

Sachant que le microprocesseur le plus rapide de tous les temps, d'après le Guiness World Record, est de $8,42938$ GHz[7], si l'on teste une possibilité par avancée d'horloge du microprocesseur, il faudra $\frac{2^{1500*8}}{8429380000}$ secondes pour tester complètement ce programme

$$\approx 8,6167 * 10^{3594} \text{ années}$$

ce qui est, en pratique, infaisable et souligne l'importance d'avoir des seeds initiales pertinentes. Au travers de cet exemple, il est clair que le fuzzing en

boite noire ne peut pas espérer tomber au hasard sur les cas problématique, il doit avoir des seeds déjà proches de ce qui pourrait causer des bugs.

Les seeds initiales, en plus d'être bien formée au début, doivent être modifiées de façon intelligente par rapport au problème. Soumettre à répétition des entrées complètement fausses ne servirait à rien : le programme le détectera et les ignorera rapidement ; ne testant que peu de celui-ci au passage [27].

Fuzzing grammatical et en boite grise

Le fuzzing en boite grise ("greybox") est la même pratique que de le fuzzing en blackbox, à une différence près. Dans ce cas-ci, on va se servir d'indications puisées directement du programme testé pour orienter la recherche [58].

Dans le cas de python, un approche possible est de compter le nombre de lignes de codes qui ont été exécutées avec une entrée donnée. Plus une entrée a permis à un grand nombre de lignes d'être exécutés, plus celle-ci va être réutilisée et mutée rapidement. Les entrées quoi n'atteignent presque aucunes lignes verront leur utilisation être nettement diminué [58].

Le fuzzer en boite grise à grammaire ajoute l'utilisation d'une grammaire pour former des entrées qui seront valides et, ainsi, explorer l'espace des entrées possibles plus efficacement en ignorant celles qui seront, dans tous les cas rejetées rapidement [59]. le défaut de cette approche est qu'elle ne teste que des entrées qui sont valides. La combiner avec une forme de mutation de seeds est, dans beaucoup de cas, intéressant et permettra d'avoir de meilleurs résultats.

Parmis les outils de fuzzing en boite grise classique, on peut notamment citer [52] :

— Peach (qui est intégré dans Gitlab a présent) [8]
— Spike [2]
— Sulley [5]

Ce type d'approche est utilisé dans plusieurs applications pratiques, tel que la recherche de failles de sécurité dans les navigateurs [31], la recherche de failles de sécurité dans les compilateurs C [56], la recherche de bug dans les implémentations de protocoles réseaux [3].

Il est possible de designer un fuzzer qui apprendra automatiquement sa grammaire via divers moyens, tel que le tracing d'exécution [33] mais ils ont un intérêt limité car il y a un risque, non-négligeable, que la grammaire apprise inclue les erreurs du programme. Cela ne sera pas systématiquement le cas mais faire attention à ce problème est nécessaire lors du design d'un système de fuzzing (pour y remédier une approche en portfolio peut être utilisée).

La limitation majeure de l'utilisation d'une grammaire pour générer des entrées valides est la grammaire en elle même. Par sa nature même elle est limitée à des entrées bien formées et ne peut pas tester les cas limites qui sont faux mais très proches d'un cas correct. Une autre limitation de cette technique héritée de la façon de fuzzer en boite noire est sa dépendance à un set de seeds initiales. Si elles ne sont pas représentatives et exhaustives des cas intéressants, le fuzzer n'aura que très peu de chance de tomber sur les cas problématiques.

Fuzzing en boite blanche

Le fuzzing dit "en boite blanche" est une variante de fuzzing plus informée. Celle-ci va commencer comme du fuzzing en boite noire mais va, au fur et la mesure de l'exécution, récolter des information et condition nécessaire pour atteindre plus de branches d'exécution (idéalement toutes mais ce n'est pas toujours possible). Les conditions récoltées lors d'une exécution seront résolues par un solveur logique et les nouvelles entrées crée par cette résolution vont être utilisées pour la prochaine exécution du code testé. ce mélange d'exécution concrète et symbolique (via la résolution de contraintes) a donné le nom de "testing concolique" à cette méthode de test. Cette méthode de tests est particulièrement efficace pour aller chercher les cas particuliers et atteindre une couverture de code particulièrement élevée.

Un exemple pratique de l'application de cette méthode est l'outil SAGE actuellement développé et maintenu par Microsoft[30]. Sage est un fuzzer qui teste en faisant de l'exécution symbolique x86 en boite blanche qui est optimisé pour gérer des stack trace énormes[27].

Cet outil est utilisé dans une des étapes finales de release des produits Microsoft et, régulièrement, découvre ce qui est appelé un "bug à millions". Les bugs à millions sont des bugs qui, s'ils passent en production, couteraient plus d'un million pour résoudre, patcher et re-déployer le code problématique[29].

SAGE étends d'autres travaux déjà réalisés dans le domaine de l'exécution concolique automatique[27] [17] [28] Cet outil n'est pas qu'un outil utilisé en recherche académique et a, actuellement, des utilisations dans l'industrie. Sa principale utilisation est par Microsoft, notamment pour le débug des OS depuis Windows 7. Il est estimé que depuis sa mise en service en 2008, a fuzzer des centaines d'applications Microsoft en continu sur des centaines de machines, sage a déjà accumulé plus de 100 années-machines d'exécution[29].

Les auteurs du solveur logique Z3 ont qualifié l'utilisation de z3 par SAGE de la façon suivante :

"Largest computational usage ever for any Satisfiability-Modulo-Theories (SMT) solver" [43]

L'utilisation la plus grande de tous les temps d'un résolveur SMT [trad]

2.1.3 Property based

Le test par propriété est une variante, plus poussée, du fuzzing décrit avant. En effet, c'est une forme de fuzzing mais les différentes entrées vont être générées à partir des propriétés qui sont définies pour chaque entrée. Initialement développé en et pour Haskell, l'utilité d'une telle forme de test, surtout dans les programmes définis strictement ou par contraintes mathématiques, s'est avérée grande [45].

Ces propriétés définie sur les entrées vont être données à un oracle qui va les traduire en entrées valides pour le programme, d'après sa définition. Ces entrées vont pourvoir être ensuite testées avec le programme en question afin de vérifier que celui-ci ne présente pas de comportement non-désirable[23].

L'avantage d'utiliser un tel système est qu'il est possible de couvrir tous les cas particuliers connus liés aux propriétés d'entrée plus facilement. En effet, avec

les contraintes définies sur les entrées, il est possible de définir une série d'entrées liées à celles-ci. Un autre avantage est qu'une fois une entrée problématique a été trouvée, il est possible d'utiliser l'oracle pour réduire cette entrée à une définition minimum afin d'aider au débug du code sous-jacent[44].

Cette méthode présente deux défauts majeurs.

Le premier est qu'elle ne permet pas, comme le fuzzing, de garantir que le programme donné est juste. Elle permet uniquement d'assurer qu'il l'est partiellement (avec les entées testées, d'où la nécessite de faire de tests exhaustifs)[44].

Le second est que le set de contraintes / propriétés à écrire pour les entrées est long a définir et peut, dans certains cas, revenir a réécrire une partie du programme lui même. Cela, dans le cas où les propriétés sont définies assez précisément pour qu'elles soient utiles [44].

2.1.4 Fault injection

L'injection de fautes est une méthode de test consistant à injecter des erreurs dans le programme testé. Il existe deux façons principales de le faire.

Introduire des erreurs au niveau hardware est une méthode qui est très peu utilisée à l'échelle du développement logiciel non dédié à du matériel spécialisé. Cette méthode consiste à, directement au niveau du matériel, provoquer des erreurs (changement de bit temporaire ou non, erreur d'accès aux diverses parties du matériel, erreurs réseaux...)[11].

Introduire des erreurs au niveau logiciel est la façon la plus répandue de faire des tests par injection de fautes. Il est possible d'en faire à divers niveaux, en boite noire et en en boite blanche. Cette méthode teste principalement la robustesse du code testé et sa capacité à récupérer des erreurs. Une autre utilisation, moins répandue, est de tester si des tests unitaires couvrent correctement le code testé.

Pour utiliser l'injection de faute pour tester la couverture de tests unitaires, une ou plusieurs fautes seront injectées dans le code. Le test réussira si les tests unitaires détectent et échouent à cause de l'erreur injectée [49].

Le test de gestion d'erreurs et de robustesse sert à vérifier que la gestion d'erreurs est correcte, que le code / le composant testé sait récupérer correctement d'un problème et de prévoir la façon dont certaines fautes vont de dérouler [11] (on peut appliquer cette méthode à des systèmes entiers [37]). Ils serviront également à déterminer la "zone d'éclat d'une erreur" : a quel point celle-ci va avoir un effet profond et grand sur le reste du composant testé. Ces tests sont particulièrement utiles pour simuler les erreurs difficiles à simuler en tests unitaires ou bien qui demandent d'émuler un système d'exploitation complet [40].

Ces erreurs peuvent être injectées a plusieurs endroits différents donc le code source initial, le binaire une fois compilé, la mémoire durant l'exécution ou encore, à un niveau plus macro, supprimer/modifier/corrompre une instance d'un déploiement distribué[11].

2.1.5 Utilisation conjointe

Toutes les techniques de fuzzing décrites précédemment peuvent être combinées pour arriver à un résultat plus efficace que leurs performances individuelles

[27]. Cette approche s'appelle faire du fuzzing hybride et est utilisée pour minimiser les problèmes liés aux différentes techniques.

Dans le cas d'une approche encore plus générale, on parlera plutôt d'une approche en portefolio [27]. Comme le fuzzing hybride, le but est de minimiser les problèmes de limitations de chaque technique tout en exploitants leurs points forts.

2.2 Tests unitaires & éducation

En 2012, suite à la constatation qu'il manquait des professeurs d'informatique, spécialement pour les 12-18 ans, l'outil Pythia a été créé. Cet outil vise à complètement enlever le professeur de la boucle de feedback tout en permettant à l'étudiant de progresser. Il inscrit dans la philosophie d'apprendre en faisant qui s'avère être très répandue au développeur de cette tranche d'âge [20].

L'université Catholique de Louvain a adopté cette plateforme et a ensuite, afin d'en dépasser les limitations, a créé une nouvelle plateforme appelée Inginious (d'autres universités ont également créé leur propre plateforme, par exemple L'Université Nationale de Colombie avec UNCODE [47]).

Inginious est une plateforme FOSS (Free Open Source Software - Un logiciel gratuit et open source) web de gradation automatique de soumission étudiant. Le code des étudiants va être compilé et exécuté dans un environnement sécurisé et, si celui-ci est correct l'utilisateur en sera informé. Dans le cas échéant, une information pertinente lui sera fournie.

Pour illustrer les développements qui ont actuellement lieu dans la correction automatique des étudiants et en raiszon de l'abondance de littérature à son sujet, la plateforme Inginious va être prise comme référence.

Cette plateforme est l'objet de nombreux développements actuellement qui ont pour but de répondre à divers challenge apparu récemment. L'un d'entre eux a été de fournir un feedback plus rapide et plus personnalisé à chaque étudiant, peu importe la quantité dont il a besoin[51].

Cette plateforme a également été utilisée pour répondre au besoin de correction des participants à un MOOC et permet à tous les étudiants d'avoir une correction adaptée et d'être certains de ou ils se situent dans leur apprentissage[22].

Elle a été adaptée pour tous types de cours, pas seulement les cours de programmation. Les cours de réseaux (et les exercices complexes liés à celui-ci) sont également intégré dans cette plateforme[15].

Elle a également été adaptée pour les enfants, revenant ainsi sur les pas de son prédécesseur tout en apportant son lot d'améliorations[54] dont entre autres l'intégration de blocks "Blocky" (une forme de développement visuel).

Parmi les améliorations récentes de ce mode d'apprentissage, on peut notamment citer l'intégration de labels afin de donner un feedback plus précis [41], l'amélioration desdits feedbacks pour être plus pertinents et avoir plus d'impact [21], ...

Cette plateforme apporte également des nouvelles opportunités au corps enseignant de connaitre le niveau des étudiants avant l'évaluation. Une variante avancée de cette pratique est l'utilisation d'intelligence artificielle pour déterminer à l'avance si un étudiant va réussir l'évaluation finale du cours[32].

Chapitre 3

Problématique

Dans le cadre de la création de nouveaux cours de programmation et d'algorithmique, le corps enseignant de l'Unamur a décidé d'utiliser la plateforme d'apprentissage Inginious comme support de développement & correction. L'utilisation de cette plateforme implique la création de nombreux tests unitaires afin de fournir un feedback pertinent, rapide et approprié aux étudiants. La taille et la complexité de cette tâche a poussé à la création de la question de recherche suivante ; Y a t il moyen d'automatiser, au moins partiellement ces tests ? Tout grain de temps pertinent, du point de vue enseignant comme du point de vue étudiant est considéré comme intéressant à creuser.

Suite à la réunion de kikoff, il a été déterminé que ce travail se concentrera sur les applications automatiques de tests unitaires suivants diverses stratégies. Il a été déterminé que toutes les stratégies de tests classiques seront évaluées. Les plus pertinentes seront implémentées et intégrées dans la plateforme Inginious. Une attention toute particulière doit être apportée à la simplicité d'utilisation (le but principal de ce travail est de gagner du temps) et la qualité du feedback fourni aux étudiants.

3.1 Sélection de stratégie de tests

Parmi tous les types de tests considérés, certains n'ont pas été retenus. Cette section va décrire pourquoi est ce que certains ont été écartés. Des innovations ont été ajoutées car permettant découvrir certains problèmes non soulevés par les autres stratégies dans le code testé.

Un autre facteur qui a été pris en compte pour la sélection des stratégies de tests a été une limite de temps. En effet, le temps de développement de ce projet n'étant pas illimité, les stratégies de tests trop complexes à mettre en place pour un retour moindre n'ont pas été retenues.

Parmi les stratégies non implémentées, il y a les tests concoliques (= une forme de fuzzing en boite blanches) qui demandent une intégration complète dans le solutionneur logique Z3 et une instrumentation complète de tout l'interpréteur Python. Il a été estimé plus sage de se concentrer sur toutes les autres techniques afin de maximiser les résultats du projet au lieu de se concentrer sur celle-là.

3.1.1 Tests par propriétés

Les tests par propriétés sont très intéressants dans le cas d'application définie très précisément qui ont besoin d'un haut niveau de fiabilité. Ce n'est pas l'approche prise par ce projet-ci, qui vide plutôt à pouvoir créer rapidement des tests avec une quantité de code et de définitions d'entrées limité. Écrire des définitions complète diminuerait fortement l'objectif d'écrire un minimum de code à l'impact de correction le plus grand possible.

Note : dans le cas de ligne écrites, il est question de l'utilisation du projet, pas d'écriture du projet de test en lui-même.

Écrire des propriétés d'entrées vagues ou imprécises pour gagner du temps ou en généricité ne feraient que rabaisser l'efficacité et l'approche de cette méthode test à un équivalent au fuzzing en boite noire. Cette technique étant considérée à part, il ne reste plus d'intérêt à faire des tests par propriété.

3.1.2 Injection de fautes

Les tests par injection de fautes sont principalement utilisées dans deux cas : le test de la qualité de tests unitaires et le test du code de récupération en cas d'erreur.

Le test de la qualité des tests unitaires n'est pas intéressant dans ce cas-ci : ils sont générés à la volée & s'adaptent dynamiquement au problème posé ce qui rend la génération de tests par injection de fautes impossible dans une majorité des cas. Les cas restant sont des cas qui ne feront que confirmer ce que a été explicitement mis en avant ou détecté par le code de génération de tests.

Le test de code de récupération est déjà ce que l'on va tester lors du fuzzing en essayant un maximum d'atteindre toutes les branches du code, dont celles de récupération. Si un cas n'est pas proprement récupéré alors que c'est demandé dans le cadre de l'exercice, cela sera repéré et remonté avec la comparaison du code testé au code de référence. En effet, le fuzzer qui a été développé compare le code testé à un code de référence et remonte une erreur si les sorties des actions ou les erreurs levées ne sont pas les mêmes.

3.1.3 Fuzzing

Durant la création de la librairie de test, certaines formes de fuzzing ont été fortement utilisées. Le fuzzing a été sélectionné dans ce cas-ci, car cette technique permet de découvrir un maximum de problèmes relativement rapidement, peu importe le code testé. Cette section va couvrir différentes techniques de fuzzing, leur avantages et leurs défauts et quelles décisions ont été prises quant à l'utilisation de celles-ci.

Fuzzing en boite noire

Le fuzzing dit "en boîte noire" ou "à l'aveugle" est un technique qui est intéressante d'appliquer quand le système testeur / de fuzzing a déjà un indice sur le type d'entrée qui pourraient être pertinentes ou qui pourraient causer des problèmes. Dans le cas de ce projet-ci, comme le code se veut très flexible et qu'il doit d'adapter à tous les exercices qui peuvent lui être soumis, cette approche n'est pas très réaliste si l'on veut être efficace et trouver les problèmes existants avec le code. L'approche du fuzzing en boite noire n'a donc pas été

sélectionnée dû à son manque de flexibilité si l'on désire obtenir des résultats pertinents rapidement.

Fuzzing en boite blanche

Le fuzzing en boite blanche et ses techniques associées (comme les tests concoliques) a été sérieusement considéré mais n'a pas su être implémenté par manque de temps et un problème de dépendance de version.

Le principal problème qui a complexifié énormément l'intégration de tests en boite blanche est la dépendance énorme que ceux-ci vont avoir sur la version précise python pour lesquels ils vont être développés. En effet, ceux-ci s'appuyant sur une instrumentation profonde et compréhensive du parseur et de l'interpréteur python, il est nécessaire de les adapter pour chaque version mineure et chaque interpréteur Python. A l'intérieur d'une version majeure de python, les mainteneurs de celui-ci garantissent que le langage ne changera pas ou que très peu, mais il n'émet aucune garantie quant au fonctionnement interne de l'interpréteur Python. Ce mode de fonctionnement pose un problème quant à la création d'une librairie pérenne de test utilisable dans plus d'un cours et que l'on peut interfacer avec les outils python que l'on veut car pas tenu par une version mineure précise.

L'autre problème qui a aidé à pousser la décision de ne pas considérer le fuzzing en boite blanche est l'intégration entre le résolveur logique Z3 et python ayant une installation et une utilisation particulièrement non triviale. Il a été estimé plus sage de concentrer le temps disponible pour le projet sur les autres types de tests, car le temps passé sur ces autres tests aura un retour supérieur au temps passé à intégrer Z3 & le lien entre Z3 et Python.

Fuzzing en boite grise

Le fuzzing en boite grise est une des approches qui a été sélectionnée dans la librairie développée. En effet, celui permet une approche hybride entre le fuzzing en boite branche en orientant les recherches tout en gardant la rapidité et la simplicité de tests inhérente aux tests en boite noire. Cette condensation des avantages des deux techniques rendent cette méthode de tests particulièrement adapté au projet actuel.

Le fuzzer développé en boite grise va se baser sur le nombre de ligne du programme qui ont été évaluées pour évaluer la qualité d'une entrée. Plus une entrée a touché un grand nombre de lignes, plus celle-ci va avoir de chance d'être sélectionnée et mutée pour le test suivant. Cette approche permet d'obtenir, très rapidement, une couverture correcte du code test, tout en passant par un maximum de branches.

Une résolution statique des expressions a également été intégrée afin de permettre au fuzzer de déduire automatiquement une série d'entrée qui pourraient être pertinentes ou permettraient d'arriver à exécuter plus de lignes du programme. le but de cette approche est d'arriver à un couverture plus grande du code testé. Celle-ci reste néanmoins limitée car, comme c'est une résolution d'expressions statique, le programme ce heure à un variant du problème d'arrêt. Cette problématique va être analysée en détails dans la partie limitations de ce document.

Afin de rendre la librairie plus facile à prendre en main, une détection automatique des arguments a été mise en place. Celle-ci se base sur les annotations d'arguments et permet de passer de code comme présent en figure 3.1 par le code plus simple présent en figure 3.2.

```
from correcteur.fuzzing.fuzz import fuzz_explicit_arguments
from correcteur.feedback.textRepporter import TestRepporter


reporter = TestRepporter()
valid_modules = ["resources.code_bidon"]

def bidon(a, b):
    res = 0
    if a > 10:
        res = a / 2
    elif b < 2:
        res += b * 3
    return (a - b + res) / (11-a)


fuzz_explicit_arguments(
        reporter,
        bidon,
        None,
        [Int(), Int()],
        valid_modules)
```

FIGURE 3.1 – Utilisation de la librairie en mode fuzzer avec le type des arguments explicitement énumérés

3.1.4 Combinaisons d'appels

Lors de la création de la librairie de tests automatiques, il a été vite mis en lumière qu'un cas en particulier, relativement facile à tester, n'était pas pris en charge avec les techniques de fuzzing classiques utilisée jusqu'à présent. En effet, dans le cadre des cours données et algorithmique, il sera, pour certains exercices, demandé aux étudiants de ne pas créer un programme complet mais uniquement une structure de données qui doit répondre à certains critères de développement. Cette structure de données peut être utilisée de beaucoup de façon différente et suivre plusieurs flux d'exécution différents ; typiquement en fonction des appels qui sont faits sur celle ci.

Par exemple, dans le cas d'une liste liée, il est possible de la manipuler de bien de façons différentes, tout en suivants certaines contraintes (dans ce cas ci, quelle doit être initialisée au début et détruite à la fin). L'ordre des ajouts, retrait et consultations n'est pas fixé et la structure doit pourvoir supporter un ordre et une quantité aléatoire de ceux-ci.

Pour accommoder ce problème une extension de la libraire de test a été

```
from correcteur.fuzzing.fuzz import fuzz
from correcteur.feedback.textRepporter import TestRepporter


reporter = TestRepporter()
valid_modules = ["resources.code_bidon"]

def bidon(a, b):
    res = 0
    if a > 10:
        res = a / 2
    elif b < 2:
        res += b * 3
    return (a - b + res) / (11-a)


fuzz(reporter,
        bidon,
        None,
        valid_modules)
```

FIGURE 3.2 – Utilisation de la librairie en mode fuzzer avec le type des arguments automatiquement détectés

ajoutée et a été concentrée du l'aspect des combinaisons d'appels sur une cible (que celle-ci soit une structure de données, un libraire ou n'importe quoi d'autre. Cela peut même porter sur une application distance sur le réseau – à condition d'adapter le runner Inginious).

Cette partie de la libraire fonctionne par étapes, divisée en actions. Les étapes seront toujours exécutées dans l'ordre déclaré et les actions au sein de celles-ci seront exécutées dans un ordre aléatoire. Plusieurs actions peuvent être exécutées au sein d'une étape. La sélection de celle-ci sera faite sur base des combinaisons générée par la librairie. Les nombres minimum et maximum d'action au sein d'une étape est réglable. Par défaut, ces nombres sont réglés à un minimum de un et un maximum de trois actions par étapes. Ces valeurs sont changeables pour chaque action au moment de leur déclaration.

Par exemple, pour que la partie combinaison de la lib teste une liste liée, une utilisation comme suit peut en être faite :

Les erreurs peuvent être détectées par la librairie de deux façons différentes :

En mode comparaisons, la librairie va tester en même temps la même combinaison d'actions sur le code de référence et le code testé. Dans l'exemple ci-dessus (fig 3.3), les appels testés sont dans le premier tableau passé aux objets "Step" et les appels de référence sont dans le deuxième tableau passé. La librairie va comparer les retours de fonctions ainsi que les erreurs remontées au code de référence lors des tests. Si le retour d'un appel de référence n'est pas exactement le même que le retour du code testé, une erreur va être logguée. Si l'exception levée par le code testé n'est pas la même que celle levée par le code de référence

```
from correcteur.steps.StepsRunner import StepRunner
from correcteur.steps.Step import Step

runner = StepRunner(stop_on_first_error=True)

runner.add_step(Step(
        [lambda: 0],
        [lambda: 0])
)
runner.add_step(Step(
        [lambda: 0, lambda: 1/0],
        [lambda: 0, lambda: 0])
)
runner.add_step(Step(
        [lambda: 0],
        [lambda: 0])
)

runner.compare_codes(reporter)
```

FIGURE 3.3 – Utilisation de la librairie en mode combinaisons
Note : pour des raisons de démonstrations une erreur a été glissée dans un des appels du code testé dans la seconde étape.

ou bien que le code testé a levé une exception et pas le code de référence une erreur va également être logguée.

En mode solo, la librairie va également exécuter uniquement le code testé, sans code de référence. Pour utiliser cette partie de la libraire sans code de référence et se baser uniquement sur les erreurs remontées pour détecter les fautes, if faut simplement passer "**None**" ou ne rien mettre comme second argument. Dans ce mode-ci, les erreurs seront détectées grâce aux exceptions lancées. Chaque exception lancée sera logguée et remontée.

Un problème récurrent qui a été noté lors du test de cette fonctionnalité de la librairie est l'inondation d'erreurs dans le logguer lorsqu'une erreur est détectée. Cela est dû au fait qu'une erreur peut, dans bien des cas, être déclenchée par de nombreuses combinaisons d'actions différentes.

Dans le cas de l'exemple cité plus haut, si une erreur et présente dans une des actions de la dernière étapes, toutes les combinaisons d'actions incluant cette action vont être comptées comme en échec et, à cause de cela, toutes celles-ci seront ajoutée dans le logger d'erreurs. Cela va générer beaucoup d'erreurs remontées à l'étudiant pour aucune valeur ajoutée. Si d'autres erreurs sont détectées, elles ne seront probablement pas vues, car noyées dans le bruit généré par cette première erreur.

La solution qui a été adoptée a été d'ajouter au *Runner* une option pour que l'évaluation cesse après une erreur détectée.

Le *Runner* est code qui va générer les combinaisons d'actions, les exécuter et les évaluer ; le *StepRunner* présent dans les figures 3.3 et 3.4 en est une

implémentation.

Dans le cas où plusieurs erreurs sont présentes dans le code, une seule sera remontée par exécution et l'étudiant devra les corriger au fur et la mesure de ses soumissions. Ce compromis a été fait, car une re-soumission n'est pas très couteuse et ne pénalise pas l'étudiant. En plus, ce retour limité à une seule erreur ne diminue que de peu la pertinence du retour donné à l'étudiant, car toutes les erreurs de son implémentation seront détectées sur les multiples soumissions de son code...

Lorsque la combinaison d'actions dans les étapes détecte une erreur, celle-ci sera remontée au code parent via un logger dédié. Celui-ci recevra, quand aucun nom n'est précisé, que le numéro de l'action qui a raté (par manque d'information disponible). Afin de permettre un feedback plus lisible et intéressant pédagogiquement aux étudiants, un paramètre optionnel, **"nice_names"** , a été créé. Celui-ci est un tableau qui contiendra un nom lisible pour l'action passée à l'étape. L'exemple de la figure 3.3 peut être modifié comme démontré dans la figure 3.4 pour inclure des noms lisibles.

L'ajout de ces noms va modifier le retour fait aux étudiant de, par exemple :

Une erreur a été trouvée après avoir exécuté les étapes suivantes : 1 -> 1 -> 2

à

Une erreur a été trouvée après avoir exécuté les étapes suivantes : init -> delete -> add

Ces noms donnés aux fonction sont liés, un à un, entre les tableaux d'actions et le tableau de noms. Le premier nom correspondra à la première action, le deuxième au deuxième et ainsi de suite.

```
from correcteur.steps.StepsRunner import StepRunner
from correcteur.steps.Step import Step

runner = StepRunner(stop_on_first_error=True)

runner.add_step(Step(
        [lambda: 0],
        [lambda: 0],
        nice_names = [
                "init"
        ])
)
runner.add_step(Step(
        [lambda: 0, lambda: 1/0],
        [lambda: 0, lambda: 0],
        nice_names = [
                "add",
                "remove"
        ])
)
runner.add_step(Step(
        [lambda: 0],
        [lambda: 0],
        nice_names = [
                "destroy"
        ])
)

runner.compare_codes(reporter)
```

FIGURE 3.4 – Utilisation de la librairie en mode combinaisons avec des noms lisibles
Note : pour des raisons de démonstrations une erreur a été glissée dans un des appels du code testé dans la seconde étape.

Chapitre 4

Contributions

La majeure partie du temps de développement a été consacrée à la création et à l'intégration d'une librairie de test automatique. Celle-ci est découpée en plusieurs éléments.

Le premier est composé des implémentations des techniques de correction que sont le fuzzing et la combinaison d'exécutions. La seconde est l'intégration de ceux-ci dans une libraire simple, contenue elle-même dans un environnent Inginious prêt à être utilisée.

Cette partie du document est dédiée à l'analyse de détails d'intégration notable ou intéressants, à l'examen de l'efficacité de a solution et a une critique de ce qui a été développé en regardant de plus près ses limitations.

4.1 Intégration

Au fil de ce projet, l'objectif principal qui a été poursuivi a été d'arriver à une intégration fonctionnelle et robuste dans Inginious dont l'utilisation est simple et claire. La petite quantité de code nécessaire à créer de nouveaux tests a également été un facteur poursuivi afin de permettre un déploiement rapide pour, par exemple, la création de nouveau cours dans la plateforme Inginious. certains points de l'implémentation et l'intégration ont reçu une attention particulière ou ont nécessité des décisions spécifiques. Cette section est dédiée à ceux-ci, aux conséquences qu'ils ont eu ainsi qu'aux modifications de design que certaines des contraintes découvertes ont impliqué.

4.1.1 Gestion des erreurs

Une grande attention a été portée sur la gestion des erreurs remontées à l'étudiant. En effet, celle-ci sont capitales pour que l'étudiant soie en mesure de comprendre ce qui n'a pas fonctionné ou ce qui est problématique et puisse y remédier.

Il est également nécessaire d'avoir une structure mise en place assez flexible pour permettre de l'étendre avec n'importe quel type d'erreur ou de besoin futur.

La structure qui a été adoptée a été une structure de logger d'erreur générique à buffer interne. L'objet python de base, tel que décrit dans le code suivant, est conçu pour être étendu.

```python
from correcteur.feedback.ErrorLog import ErrorLog

class ErrorReporter:
    def __init__(self):
        self.errors = []

    def add_error(self, error: ErrorLog):
        self.errors.append(error)

    def get_output(self) -> [str]:
        raise NotImplementedError

    def get_text_output(self) -> str:
        res = ""

        if len(self.get_output()) != 0:
            for line in self.get_output():
                res += line
                res += "\n"
        return res
```

Il a été considéré comme plus intéressant d'avoir une structure générique améliorable facilement en divers loggers spécialisés plutôt qu'un seul logger qui serait difficile à maintenir, à étendre, à intégrer et à débugger.

Pour des facilités d'utilisation de la librairie, un logger textuel simple, étendant l'objet "ErrorReporter" de base est fourni. Celui-ci est visible dans le code suivant.

```python
from correcteur.feedback.errorRepporter import ErrorReporter


class TestRepporter(ErrorReporter):

    def get_output(self) -> [str]:
        res = []
        for error in self.errors:
            res.append(
                "[{method}] {msg} (Error: \"{error}\")".format(
                    error=error.error,
                    method=error.methode,
                    msg=error.message))
        return res
```

Ce rapporteur, très simple, se content de mettre l'erreur levée dans une string formatée de façon intelligible pour l'étudiant. Son gros défaut est qu'il compte sur le fait que les erreurs qui lui sont reportée sont déjà complètement utilisables comme feedback et ne fait aucun traitement supplémentaire des inputs qui lui sont données afin de les améliorer ou de les rendre plus compréhensibles.

Un exemple type de ce logger dans un des algorithmes de détection de faute peut être observé dans le code suivant.

```
from correcteur.feedback.ErrorLog import ErrorLog
from correcteur.feedback.errorRepporter import ErrorReporter

# testing fait et les erreurs detectee sont
# collectees dans le tableau "fail"

for i in fail:
        reporter.add_error(ErrorLog("fuzzer", i[1],
                "the fuzzer broke the" \
                " code  (inputs: {})".format(i[0])))
```

Comme vu dans la figure précédemment mentionnée, le logger est passé comme argument à chaque exécuteur. Cela permet, si nécessaire, de passer un logger personnalisé / spécialisé / adapté différent à chaque structure afin de personnaliser un maximum la remontée d'erreur et donner un retour le plus pertinent possible à l'étudiant.

4.1.2 Ajout d'un type de donnée

Comme pour la gestion des erreurs, l'ajout d'un nouveau type de donnée dans le fuzzer a été conçu pour être très simple à faire. Un type de donnée est représenté par un objet qui a plusieurs rôles quant à la gestion de la vie de ce type dans le fuzzer. C'est celui-ci qui va :

- les créer de façon aléatoire
- les muter
- sélectionner les cas spéciaux à partir des tokens statiques trouvés dans le code source
- les créer à partir des cas spéciaux

Pour en créer un nouveau, il va simplement être nécessaire d'étendre la classe input (*correcteur/fuzzing/input/input.py*) au sein du même module. Comme mentionné précédemment, la classe input contient tout ce qui est nécessaire pour permettre au fuzzer de travailler avec le type de donnée représenté par ses classes enfants.

```
class Input:
    def __init__(self):
        pass

    def integrate_by_type(self, candidates):
        """
        Will take a list of candidates as an input
        and integrate the compatibles one as internal
        base seeds
        :param candidates: the set of potential candidates
        """
        raise NotImplementedError

    def get_special_cases(self):
        """
        :return: an array of seed for this type of
```

```
            runner that represent base special cases
        """
        return []

    def get_random(self):
        """
        :return: a new, completely random seed for
                 this input
        """
        raise NotImplementedError

    def can_mutate(self):
        """
        Will return if the type is able to be mutated
        """
        return True

    def mutate(self, value):
        """
        :param value: the base input
        :return: a new , mutated by a bit, value
        """
        raise NotImplementedError

    def is_valid_type(self, candidate):
        """
        :param candidate: the python object that is
                 going to be analysed
        :return: True if the given object match the
                 type of this one
        """
        raise NotImplementedError
```

Une fois le nouveau type créé, il ne reste plus qu'à l'ajouter à la fonction qui fait la comparaison entre les annotations présentes sur la fonction testée et les types tels qu'implémentés ici. Faire ce rajout est trivial car, comme montré dans le code suivant, la fonction est très simple dans sa conception.

```
def get_type_from_str(name):
    if name == "int":
        return Int()
    if name == "float":
        return Float()
    if name == "bool":
        return Bool()
    if name == "str":
        return Str()

    if name == "List[int]":
        return List(internal_type=Int())
```

```
    if name == "List[float]":
        return List(internal_type=Float())
    if name == "List[bool]":
        return List(internal_type=Bool())
    if name == "List[str]":
        return List(internal_type=Str())

    raise NotImplementedError()
```

Note importante à propos de cette fonction : le type "List" vu dans celle-ci n'est ***pas*** *le type list primitif de python mais bien un type de donnée implémenté comme les autres.*

Les types composites (composé de plusieurs sous-types) sont supporté, comme, par exemple, l'implémentation du type "List" le démontre. Ceux-ci marchent de la même façon que leurs homologues simples, à la différence près qu'ils vont déléguer certaines taches et informations aux types qu'ils contiennent. Pour qu'un type composé soit pris en compte, il est nécessaire d'ajouter toutes ses formes dans la fonction décrite plus haut pour qu'il soit possible de l'utiliser automatiquement au bon endroit lorsque l'utilisation d'annotation de types est utilisée.

4.1.3 Intégration dans Inginious

Pour faciliter une intégration rapide et sans problèmes dans la plateforme Inginious, un fichier Docker pour créer un conteneur de vérification Inginious a été créé. Une fois ce conteneur créé sur la machine hébergeant Inginious, il suffit de relancer Inginious puis d'aller sélectionner l'environnement "python3-correcteur" dans la tâche.

Ce fichier de construction docker se base sur l'environnent python 3 de base et ajoute la librairie développée dans le cadre de ce projet sur le path Python afin de permettre une importation aussi simple, par exemple, que ceci :

```
from correcteur.fuzzing.fuzz import fuzz
```

Le dockerfile est composé comme suit :

```
FROM ingi/inginious-c-python3
LABEL org.inginious.grading.name="python3-correcteur"


RUN pip3 install astunparse
RUN mkdir -p /python
COPY correcteur /python/correcteur
ENV PYTHONPATH="/python:\${PYTHONPATH}"
```

Pour aider à l'intégration du projet dans Inginious, un cours d'exemple a été créé et contient un exemple des quatre modes possibles :

- fuzzer avec fonction de validation
- fuzzer sans fonction de validation
- combinateur avec code de référence
- combinateur sans code de référence

ces quatre modes donnent leur feedback de la même façon : écrire le feedback sur la sortie' standard et se servir de cet output comme message de retour. Ce comportement est, bien sur, hautement personnalisable et est adaptable facilement aux modèles personnalisés de feedback Inginious.

4.2 Efficacité

Ce projet a été tune pour être réalistement utilisable dans le contexte de l'enseignement. Pour ce faire, certaines concessions ont été faites et certaines décisions arbitraires ont été prises. Celles ci-sont l'objet de cette section.

4.2.1 Type d'erreurs détectées

Les deux approches utilisées sont particulièrement adaptées pour trouver certains types de problèmes. Celles-ci couvrent une certaines parties des erreurs possibles mais pas toutes.

Le fuzzer sera particulièrement bon pour trouver des cas particuliers non supporté sur les entrées. De par sa nature et le fait qu'il est, dans ce cas-ci, fais pour prendre comme entrées de base les nombres hard-codés du programme ainsi que les cas particuliers de chaque type, il va être très efficace pour réussir à déceler les bugs d'entrées non saines. L'utilisation principale faite de celui-ci pour trouver les problèmes avec les parties de programme en contact avec les entrées utilisateurs (comme, par exemple le "Security Development lifecycle" créé par Microsoft [34]) abonde dans ce sens.

L'algorithme de combinaison d'appels va être particulièrement utile pour trouver des combinaisons d'action qui vont créer un état interne du programme indésirable. Tous les états problématiques ne seront pas trouvés, seuls ceux qui résultent en une erreur détectable par le programme de test. Celui-ci reste limité par son ignorance des paramètres et sa concentration sur les appels (les paramètres sont fixés au travers de tous les appels).

Les problèmes non trouvés peuvent être multiples.

Les bugs qui ont besoin, pour être déclenchés de combinaisons d'appels et de paramètres spécifiques vont être très difficiles à déceler. Globalement, plus un problème présent dans un programme a besoin d'une grande combinaison d'appels ou d'un nombre élevé de paramètre pour arriver, plus difficilement il sera décelé par les approches de test automatique décrites dans ce document. Une façon de palier en partie à ce problème de rajouter certaines analyses statiques et des conventions fortes pour éviter que ces situations problématiques complexes puissent apparaître.

Les problèmes qui, pour être détectés ou apparaitre dépendent d'évènement externe non anticipé sont également plus difficiles à déceler avec les approches prises ici. L'environnent étant fort contrôlé, ajouter des tests par l'injection de fautes ne serait pas fort intéressant. Il est, en effet, possible dans ce cas-ci, d'empoisonner les interfaces avec ces évènements extérieurs pour simuler le problème (en pratique, cela peut être fait avec le combinateur). Une autre approche possible est d'utiliser de l'injection de fautes.

4.2.2 Valeurs par défaut

Un des choix arbitraire qu'il a fallu faire sont toutes les valeurs par défaut utilisées. Celles ci sont très importantes car, si bien réglées, permettent d'utiliser plus facilement le projet en devant développer moins, ce qui est un des objectifs de ce projet.

Ces valeurs par défauts ont été trouvées empiriquement en utilisant un projet relativement petit mais pas minuscule. Cette taille décidée arbitrairement s'est basée sur ce que je considérais comme une taille d'exercice de base en algorithmique (une implémentation partiellement fausse de Dijkstra a été utilisé). La machine sur laquelle ces tests ont été faits étant relativement puissante (cpu : AMD ryzen 7 5800x), cet effet a également été pris en compte en divisant le résultat obtenu par la fréquence de la machine hôte et multipliant le chiffre obtenu par 2.5Ghz (un cpu de serveur moyen de gamme actuel, par exemple le AMD EPYC 7502, atteint ces performances). Ce nombre a finalement été arrondi vers le bas pour rajouter un peu de marge.

4.3 Limitations

Cette librairie n'est pas parfaite.

Elle a été conçue de façon à optimiser plusieurs choses.

Le temps de correction doit être assez court pour qu'au moment de son retour, la réponse soit encore pertinente. Les tests automatiques ne sont pas omniscients et peuvent avoir des zones aveugles en fonction de la situation. Son développement a été étalé sur une période limitée et les approchées sélectionnées ont tenu compte de ce facteur.

Tous ces éléments combinés ont pour conséquence que cette librairie, bien que conçue pour tirer le meilleur de certaines tactiques de détection de problèmes, n'est pas la solution parfaite ultime mais une approximation qui se veut le plus proche de celle-ci.

Certaines des limitations sont traitées plus en profondeur au fil de la section suivante.

4.3.1 Tests d'objets génériques

Une proposition qui a été écartée qui aurait pu, dans d'autres circonstances que celle ce projet être utile, est la création à la volée d'objets correspondant à ce qui est attendu par le code testé en se basant sur les erreurs retournées par celui-ci. Le "duck typing" de python permet de faire cela sans trop de difficulté (Deux objets différents en python ayant les mêmes signatures de méthodes et les mêmes noms de variables sont considérés comme étant de même type. Pour être plus précis, les deux types sont considérés comme identiques.).

Le problème est que, au lieu de créer à la volée l'objet qui est *censé* être passé au programme, il sera construit systématiquement l'objet qui correspond parfaitement à ce que le programme accepte, que ça soit juste ou non. Ces objets créés ne correspondront pas à ce qui va, lors de l'utilisation du programme, être réellement passé à celui-ci. Cette technique valide uniquement que le programme accepte des objets qui sont parfaits pour ce qu'il est non ce qu'il *devrait être*.

4.3.2 Résolution des branches

Résoudre statiquement toutes les expressions présente dans le programme testé permet au fuzzer de partir sur une base d'entrées saines et pertinentes.

Les plus faciles à détecter sont les nombres magiques présents dans le programme (un nombre magique est, dans le contexte d'un programme, une constante basée sur rien / peu de choses que le développeur a utilisé). Les expressions simples suivent rapidement derrière car un algorithme de parcours et une implémentation de calculatrice saura les résoudre.

Un des problèmes qui va très vite être rencontré lorsque de plus grandes expressions vont être considérées, est qu'il est impossible de statiquement affirmer si une boucle quelconque dans un programme va se finir et si oui, si c'est dans un temps raisonnable. Ce problème peut être vu comme une application du problème d'arrêt [16] dans lequel la boucle est un programme inclus dans un programme plus grand. Cette idée qu'une boucle peut être une application du problème d'arrêt peut être étendue à n'importe quel groupe d'instruction, ce qui rend l'exécution d'un analyseur statique impossible pour les expressions plus complexes comprenant des structures de contrôle et non uniquement des mathématiques simples.

Un parseur de code générique a été développé pour permettre de résoudre ces instructions facilement. Afin de rester compatible avec un maximum de versions de python, il n'assume que très peu de choses à propos de l'environment python dans lequel il est exécuté. Ceci est accompli avec une récursivité générique basée sur le parseur python.

Celui-ci peut être trouvé dans le projet dans le dossier "correcteur/fuzzing/-token", nommé "*universal_magic_token_finder.py*".

4.3.3 Détection automatiques du type d'arguments

Une détection automatique, sans les annotations, du type d'argument qu'une fonction doit recevoir a été envisagé mais a été mis de côté à cause du peu d'avantage que cette technique apportera pour sa complexité et sa non-fiabilité.

Plusieurs problèmes, certains liés à la façon dont Python est conçu vont empêcher cette idée de fonctionner proprement.

Le fait qu'une variable en Python puisse être de n'importe quel type et changer plusieurs fois de type au cours de sa vie va complexifier énormément la détection du type de l'argument.

Les fonctions en python n'ayant pas le type de retour inclus obligatoirement dans la signature de la fonction, il est impossible de déduire automatiquement le type de retour de la fonction sans devoir remonte à travers le code de cette fonction récursivement. Pour palier à cela, il sera nécessaire de contraindre logiquement résolution du type d'argument, ce qui impliquerait de prédire certaines informations sur la fonction. Une des conséquences du typing dynamique de Python est qu'une des choses qui devrait être déduite serait de savoir si une branche serait explorée ou non. Par exemple, si le résultat d'une condition complexe est passée à la fonction de la figure 4.1, déterminer statiquement le type de la sortie devient impossible sans avoir résolu la valeur du bool d'entrée, sachant que celui peut rester indéfini tant que le programme n'est pas exécuté et peut varier d'une exécution à l'autre. Déterminer si une branche va être atteinte ou

pas est une extension du problème d'arrêt [16] : au lieu de chercher si on va atteindre une branche particulière (celle qui mène à l'arrêt du programme), on va cherche à atteindre n'importe quelle autre branche en particulier.

```
def problematique(a: bool):
        if a:
                return 1.0
        return "deux"
```

FIGURE 4.1 – Programme qui poserait un problème à la détection automatique d'argument

Certaines fonctions étant génériques et adaptées à plusieurs types, cela rend, en plus des autres problèmes cités, la détection du type des arguments d'une fonction inutile, car il n'est pas passible de déterminer automatiquement si c'est voulu et donc si le problème trouvé avec un type particulier est pertinent et bon à signaler.

4.3.4 Profondeur limitée

La partie de la librairie dédiée au combinateur est fortement limitée par le nombre d'entrées qui lui sont fournies ainsi que la profondeur maximale l'algorithme va explorer. Le terme profondeur est utilisé ici, car on peut visualiser les différentes combinaisons comme un arbre de possibilités. Plus le nombre de combinaison maximum d'actions par étape est grand plus la profondeur de l'arbre de possibilités généré va être élevé). Comme montré sur la figure 4.2, la croissance de cet arbre étant exponentielle, la taille en devient vite problématique et incalculable.

θ est le nombre d'étapes
ϕ est le nombre d'actions disponibles par étape
α est le nombre maximum d'actions exécutables par étape

Taille de l'arbre $= f(\theta, \phi, \alpha) = \theta * \phi^{\alpha}$

θ \ α	1	3	5	11	17
1	5	125	3125	48828125	762939453125
3	15	375	9375	146484375	2288818359375
5	25	625	15625	244140625	3814697265625
11	55	1375	34375	537109375	8392333984375
17	85	2125	53125	830078125	12969970703125

Dans le tableau, la supposition est faite que ϕ vaut 5. Un ϕ de 5 a été sélectionné car c'est raisonnablement atteignable dans une utilisation normale du projet. Dans la formule et le tableau la limite maximale du nombre d'étapes n'est pas considérée.

FIGURE 4.2 – Taille de l'arbre de combinaisons d'actions

Une solution a été apportée à ce problème sous la forme d'une limite globale de longueur de combinaison générée. Toutes les combinaisons plus longues ce quelle ci ne seront pas générées.

Dans l'implémentation actuelle elles ne seront pas juste écartées, elles ne sont pas du tout générées pour gagner en rapidité et éviter de perdre du temps à créer des entrées inutiles.

4.3.5 Combinaisons de techniques

Une limitation connue de la partie combinatoire du projet est le manque de paramétrages des actions. Chaque action ne peut, en effet, pas avoir de paramètre ou bien, uniquement des paramètres statiques. Une des solutions possible aurait été de combiner un fuzzer avec cette partie du projet pour arriver à une solution couvrant tout.

Cela a été envisagé mais n'a pas été fait pour une question de réalisme. Il n'est pas envisageable de, réalistement, fuzzer toutes les étapes de toutes les combinaisons d'exécution d'actions : cela amènerait une complexité temporelle beaucoup trop élevée et ne serait pas utilisable dans le cadre du projet. Les soumissions des étudiants ont besoin d'être rapidement évaluées et un compromis entre complétude et rapidité a été fait ici. Sans celui-ci l'étudiant aurait un retour extrêmement précis... après plusieurs années de traitement de la soumission.

4.3.6 Parallélisme

Une autre limitation du projet, pas liée au projet en lui-même mais à Python est le manque de parallélisme. En effet, l'implémentation la plus classique de l'interpréteur Python, CPython, utilise GIL (Global Internal Lock) sur tous les objets python[1].

Ce lock global interne est utilisé pour protéger tous objets Python et empêcher les accès concurrents à ceux ci. La seule forme de concurrence que cet interpréteur python supporte est l'entrelacement de threads au sein d'un seul et unique processus.

Les seuls gains possibles sont des gains possibles sur les temps d'I/O, ce qui est relativement peu présent dans les tests visés par ce projet (étant du code pur, en général sans accès réseau). Même dans ce cas-là, les gains seront très vite nullifiés, car le temps perdu à l'orchestration des threads deviendra plus grand que le temps gagné par l'utilisation des threads. Cela implique que, même en créant une multitude de threads, l'efficacité des tests n'en sera que, au mieux, très peu augmenté et dans la majorité des cas, diminué.

Une autre possibilité est d'utiliser plusieurs processus différents mais, de nouveau, comme l'interpréteur Python a un GIL[6] et ne supporte pas le parallélisme cette option est compliquée à mettre en place. La seule solution est de lancer plusieurs interpréteurs en même temps et d'utiliser une technique d'IPC (Inter-Process Communication) pour les synchroniser. Cette solution, très lourde, est plateforme dépendante et est donc très peu intéressante dans le cas de ce projet qui se veut rapide, léger et efficace.

4.3.7 Limite de temps

Limiter dans le temps l'exécution d'un programme python est très compliqué à faire de façon générique sans être extrêmement dépendant des plateformes, OSs et version utilisées.

Pour interrompre le code en cours de test, si celui-ci ne finit pas, il est nécessaire de le faire depuis un autre thread. Le fait que l'implémentation la plus utilisée (et de référence) de python (CPython) ne supporte que très faiblement le parallélisme (le GIL en est un exemple de ce faible support [1][6]) et que l'orchestrateur du système d'exploitation ne gère pas les threads au sein des processus, va résulter au fait qu'il va être nécessaire, pour développer un timeout fiable de passer par un autre programme, synchronisé par une technique d'IPC (Inter-Process Communication) pour arrêter un thread trop long. Ce processus étant très plateforme dépendant et comme Inginious dispose déjà d'un mécanisme de timeout, il a été décidé de ne pas implémenter de mécanisme d'arrêt au sein de la librairie. Celui-ci aurait été intéressant pour pouvoir donner un feedback plus précis à l'étudiant quant à pourquoi et dans quelles circonstances est ce son code a fait une boucle infinie.

Chapitre 5

Conclusion

Chapitre 6

Bibliographie

[1] 8.1 Thread State and the Global Interpreter Lock — web.archive.org. https://web.archive.org/web/20080914102629/http://docs.python.org/api/threads.html. [Accessed 06-May-2022].

[2] Fuzzer Automation with SPIKE - Infosec Resources — resources.infosecinstitute.com. https://resources.infosecinstitute.com/topic/fuzzer-automation-with-spike/. [Accessed 18-Apr-2022].

[3] Github - aflnet/aflnet : Aflnet. https://github.com/aflnet/aflnet. [Accessed 18-Apr-2022].

[4] GitHub - Netflix/chaosmonkey : Chaos Monkey is a resiliency tool that helps applications tolerate random instance failures. — github.com. https://github.com/netflix/chaosmonkey. [Accessed 01-May-2022].

[5] GitHub - OpenRCE/sulley : A pure-python fully automated and unattended fuzzing framework. — github.com. https://github.com/OpenRCE/sulley. [Accessed 18-Apr-2022].

[6] GlobalInterpreterLock - Python Wiki — wiki.python.org. https://wiki.python.org/moin/GlobalInterpreterLock. [Accessed 06-May-2022].

[7] Highest clock frequency achieved by a silicon processor. https://www.guinnessworldrecords.com/world-records/98281-highest-clock-frequency-achieved-by-a-silicon-processor. Guinness World Records.

[8] Integrating security into your DevOps Lifecycle — peachfuzzer.com. http://www.peachfuzzer.com/. [Accessed 18-Apr-2022].

[9] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. Automated unit test generation for classes with environment dependencies. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, September 2014.

[10] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, May 2014.

[11] D. Avresky, J. Arlat, J.-C. Laprie, and Y. Crouzet. Fault injection for formal testing of fault tolerance. *IEEE Transactions on Reliability*, 45(3) :443–455, 1996.

[12] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3) :1–39, May 2019.

[13] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. *ACM SIGPLAN Notices*, 52(6) :95–110, 2017.

[14] Netflix Technology Blog. The netflix simian army, Sep 2018.

[15] O. Bonaventure, Q. De Coninck, F. Duchêne, A. Gégo, M. Jadin, F. Michel, M. Piraux, C. Poncin, and O. Tilmans. Open educational resources for computer networking. *ACM SIGCOMM Computer Communication Review*, 50(3) :38–45, jul 2020.

[16] Leslie Burkholder. The halting problem. *ACM SIGACT News*, 18(3) :48–60, 1987.

[17] Cristian Cadar and Dawson Engler. Execution generated test cases : How to make systems code crash itself. In *International SPIN Workshop on Model Checking of Software*, pages 2–23. Springer, 2005.

[18] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, May 2011.

[19] Gul Calikli and Ayse Bener. Empirical analysis of factors affecting confirmation bias levels of software engineers. *Software Quality Journal*, 23(4) :695–722, sep 2014.

[20] Sébastien Combéfis and Vianney le CLÉMENT de SAINT-MARCQ. Teaching programming and algorithm design with pythia, a web-based learning platform. *Olympiads in Informatics*, 6, 2012.

[21] Guillaume Derval, Anthony Gego, Pierre Reinbold, Benjamin Frantzen, and Peter Van Roy. Génération de feedback enrichis pour la plateforme inginious.

[22] Guillaume Derval, Anthony Gego, Pierre Reinbold, Benjamin Frantzen, and Peter Van Roy. Automatic grading of programming exercises in a mooc using the inginious platform. *European Stakeholder Summit on experiences and best practices in and around MOOCs (EMOOCS'15)*, pages 86–91, 2015.

[23] George Fink and Matt Bishop. Property-based testing. *ACM SIGSOFT Software Engineering Notes*, 22(4) :74–80, July 1997.

[24] Justin E. Forrester and Barton P. Miller. An empirical study of the robustness of windows nt applications using random testing. In *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4*, WSS'00, page 6, USA, 2000. USENIX Association.

[25] Gordon Fraser and Andrea Arcuri. A large-scale evaluation of automated unit test generation using EvoSuite. *ACM Transactions on Software Engineering and Methodology*, 24(2) :1–42, December 2014.

[26] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated unit test generation really help software testers ? a controlled empirical study. *ACM Transactions on Software Engineering and Methodology*, 24(4) :1–49, September 2015.

[27] Patrice Godefroid. Fuzzing. *Communications of the ACM*, 63(2) :70–76, January 2020.

[28] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart : Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[29] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE : Whitebox fuzzing for security testing. *Communications of the ACM*, 55(3) :40–44, March 2012.

[30] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[31] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, 2012.

[32] Robin Hormaux and Siegfried Nijssen. Predire la reussite d un etudiant en analysant les donnees d inginious : une approche par machine learning.

[33] Matthias Hoschele and Andreas Zeller. Mining input grammars with autogram. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 31–34. IEEE, 2017.

[34] Michael Howard and Steve Lipner. *The security development lifecycle*, volume 8. Microsoft Press Redmond, 2006.

[35] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7) :385–394, July 1976.

[36] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. Efficient unit test case minimization. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE '07*. ACM Press, 2007.

[37] Rakesh Kumar Lenka, Sarthak Padhi, and Kabita Manjari Nayak. Fault injection techniques - a brief review. In *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*. IEEE, oct 2018.

[38] 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*. ACM, nov 2014.

[39] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, May 2007.

[40] Paul D. Marinescu and George Candea. Efficient testing of recovery code using fault injection. *ACM Transactions on Computer Systems*, 29(4) :1–38, December 2011.

[41] Olivier Martin and Olivier Bonaventure. Improving inginious : labeling mechanism to better identify difficulties of students.

[42] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. Jseft : Automated javascript unit test generation. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, 2015.

[43] Leonardo de Moura and Nikolaj Bjørner. Z3 : An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[44] Manolis Papadakis and Konstantinos Sagonas. A PropEr integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang - Erlang '11*. ACM Press, 2011.

[45] Zoe Paraskevopoulou, Cătălin Hriţcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. Foundational property-based testing. In *Interactive Theorem Proving*, pages 325–343. Springer International Publishing, 2015.

[46] Xiao Qu and Brian Robinson. A case study of concolic testing tools and their limitations. In *2011 International Symposium on Empirical Software Engineering and Measurement*. IEEE, September 2011.

[47] Felipe Restrepo-Calle, Jhon J Ramírez-Echeverry, and Fabio A González. Uncode : Interactive system for learning and automatic evaluation of computer programming skills. *EDULEARN18 Proceedings*, 1 :6888–6898, 2018.

[48] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. Automated unit test generation during software development : a controlled experiment and think-aloud observations. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, July 2015.

[49] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Ysskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin. FlAT – fault injection based automated testing environment. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'*. IEEE.

[50] Koushik Sen. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE '07*. ACM Press, 2007.

[51] Thomas Staubitz, Ralf Teusner, and Christoph Meinel. Towards a repository for open auto-gradable programming exercises. In *2017 IEEE 6th International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*. IEEE, dec 2017.

[52] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing : brute force vulnerability discovery*. Pearson Education, 2007.

[53] Kunal Taneja and Tao Xie. Diffgen : Automated regression unit-test generation. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 407–410, 2008.

[54] Florian Thuin and Olivier Bonaventure. Inginious for kids.

[55] Xinyu Wang, Jun Sun, Zhenbang Chen, Peixin Zhang, Jingyi Wang, and Yun Lin. Towards optimal concolic testing. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, May 2018.

[56] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.

[57] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2021. Retrieved 2021-10-26 15 :30 :20+02 :00.

[58] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. Greybox fuzzing. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2022. Retrieved 2022-01-23 17 :13 :33+01 :00.

[59] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. Greybox fuzzing with grammars. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2022. Retrieved 2022-01-11 09 :26 :47+01 :00.

[60] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4) :366–427, dec 1997.

Annexes