Analyse de la Propagation basée sur les Graphes Logiciels et les Données Synthétiques

Titre original: Propagation Analysis based on Software Graphs and Synthetic Data

Vincenzo Musco

1 Introduction

A notre ère, les systèmes informatiques sont omniprésents ; un large spectre de tâches sont aujourd'hui accomplies par des machines. En vue de réaliser ces tâches, ces machines doivent se voir dictées ce qu'elles doivent concrètement faire : il s'agit là de la tâche du logiciel. Les développeurs sont ainsi les acteurs principaux dans la création desdits logiciels : ils décrivent les opérations que la machine doit accomplir par le biais du code source.

Malheureusement, ces développeurs sont des êtres humains et ceux-ci sont connus pour être faillibles. Ainsi, il n'y a pas d'exception pour les logiciels informatiques : ils contiennent des erreurs. Ces erreurs font ainsi partie intégrante du logiciel et peuvent aboutir à divers scénarios d'exécution indésirés comme par exemple, l'arrêt injustifié du programme, des composants d'interface graphique ne répondant pas et même le plantage complet du système.

Le code source d'un programme pouvant s'étaler sur plusieurs milliers de lignes de code, découpées en une multitude de fichiers, la tâche de débogage peut rapidement devenir un vrai casse-tête pour le développeur : l'erreur peut se trouver n'importe où dans le code source. Un large éventail d'erreurs peuvent ne pas être détectables par le compilateur et resteront dormantes jusqu'à ce que le programme soit exécuté. Un exemple classique d'erreur dormante est le cas où une méthode est appelée sur un objet non instancié, résultant dans la classique exception nulle en Java. De plus, lorsqu'un changement est introduit dans un programme, un ou plusieurs effets de bord peuvent se manifester : un changement C_1 dans le code source peut avoir des conséquences sur n'importe quel autre partie du code dépendant de C_1.

Quand un développeur ou un testeur constate une exécution non désirée du logiciel, la tâche naturelle qui suit est en général la correction du code source en vue de supprimer cette exécution non désirée. Cependant, un pré-requis inévitable est qu'il faut identifier l'endroit du code source à modifier. Malheureusement, cette tâche peut parfois être plus compliquée que la correction en elle-même.

Face à l'omniprésence de ces fautes et de la difficulté à identifier le point dans le code source étant à l'origine de celles-ci, un nombre important d'outils ont été mis en place pour assister au débogage et à la correction des erreurs. Cela commence tôt dans le processus de développement en procédant à une bonne spécification et documentation du projet, notamment par l'utilisation de commentaires. Ceux-ci ne sont ni compilés, ni exécutés, mais sont là pour aider le développeur à se souvenir des raisons et des implications de ses choix de développement. De plus, utiliser un système de log peut grandement aider le développeur à comprendre un comportement non désiré du programme.

Le test logiciel est une approche populaire au sein de laquelle le développeur annexe à son code source des cas d'exécution simples décrivant le comportement attendu. Ces tests sont ensuite exécutés à de multiples reprises durant le processus de développement de telle sorte à ce qu'une erreur introduite soit détectée rapidement.

Les systèmes de rapports de bugs proposent également à des personnes extérieures au développement du système de reporter et de discuter d'éventuels comportements non désirés qu'ils auraient pu rencontrer durant leur utilisation du programme. Grâce à des systèmes de gestion de versions tels que Git ou Subversion, les développeurs peuvent facilement collaborer sur des projets et bénéficier d'un grand nombre d'outils tels que le versionnage, c'est-à-dire l'historique complet des changements qui ont été apportés au logiciel.

Un grand nombre de techniques d'assistance face aux fautes ont également été proposées par les chercheurs dans le domaine du génie logiciel. Ces techniques ont pour but d'aider les développeurs à détecter, trouver et corriger les fautes présentes dans le logiciel.

Certains outils ne sont pas seulement liés aux fautes, mais ont pour but d'assister le développeur avec des tâches fastidieuses et qui pourraient conduire à des erreurs. Un exemple est le cas du "refactoring" qui permet à un développeur de gérer toutes les contraintes liées à une opération (déplacer, renommer, etc.) dans un fichier de code source. Le refactoring permet de s'assurer qu'à terme de l'opération, la cohérence au sein du programme est bel et bien préservée.

Ces approches utilisent généralement tous les artefacts du logiciel mis à disposition. Ceux-ci incluent notamment le code source en lui-même, mais aussi tous les éléments cités précédemment : documentation, rapports d'erreurs, tests logiciels, gestionnaire de version, etc.

Pour conclure, la détection, l'identification et la correction des fautes logiciels sont des tâches demandant énormément de temps, fastidieuses et qui peuvent s'avérer compliquées. Un changement, même minime, du code source peut avoir pour conséquence de rendre d'autres parties du code inopérantes. Pour toutes ces raisons, un grand nombre d'outils sont proposés pour assister les développeurs dans leurs tâches quotidiennes. Avec le temps qui passe, les systèmes deviennent de plus en plus complexes, ces outils vont devenir de plus en plus appréciés.

1.1 Problèmes

Dans cette thèse, nous nous concentrons principalement sur des problèmes liés à l'analyse d'impact. Celle-ci consiste en l'étude de la propagation, via les appels de méthodes, les variables, etc d'un changement quelconque du code source au sein de logiciels dont l'impact aura une influence sur une autre partie du logiciel.

Cette problématique de propagation peut être vue selon deux points de vue : la propagation ante-mortem et post-mortem. La propagation ante-mortem consiste en la capacité à rapporter les potentiels impacts d'un changement au développeur directement lorsque celui-ci est introduit dans le code source, sans nécessité une quelconque exécution de celui-ci. Dans cette optique du problème, nous parlons d'un problème d'analyse d'impact au changement (appelé CIA, change impact analysis en anglais).

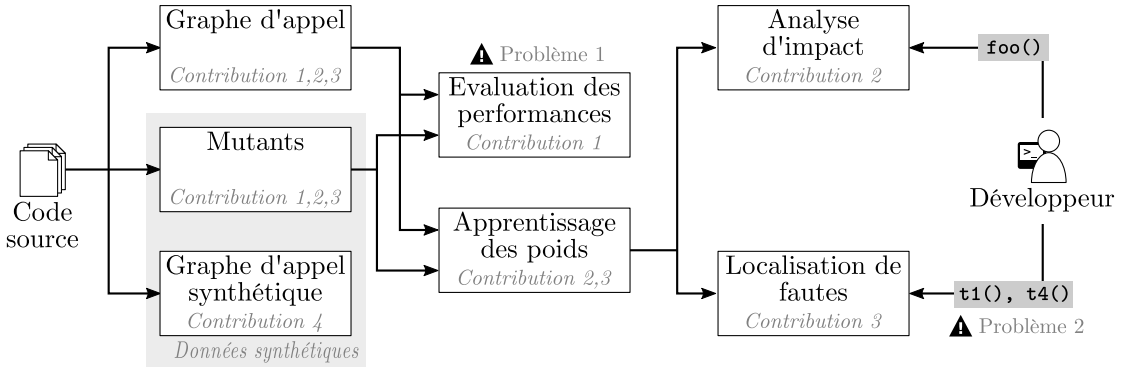

FIGURE 1 – Problèmes et contributions abordées dans cette thèse.

Problème 1

Pour les techniques d'analyse d'impact au changement existantes, il n'existe pas d'approche d'évaluation claire permettant de calculer et de comparer les performances de celles-ci. Par performance, nous entendons la capacité pour ces techniques de localiser avec exactitude les éléments qui seront réellement impactés.

La propagation post-mortem consiste à rapporter un ensemble d'éléments (c'est-à-dire des fautes potentielles) qui sont responsables d'un certain impact (aboutissant au dysfonctionnement du programme ou du système). La détection de la faute a lieu après que celle-ci se soit manifestée, donc après que le code source aie été exécuté. Dans cette optique du problème, nous parlons d'un problème de localisation de fautes (appelé LF, fault localization en anglais). En essence, le processus de la localisation de fautes essaye de capturer les liens de causes existant entre les différent éléments du code source.

Problème 2

La plupart des techniques de localisation de fautes ne considèrent pas le programme dans son ensemble, ils mettent l'accent sur un ensemble d'éléments rapportés par leur approche sans considérer la façon dont ces éléments interagissent dans l'ensemble du logiciel.

Dans cette thèse, nous nous attaquons à ces problèmes sur base de deux intuitions fondamentales. La première est que *les graphes sont des structures de données particulièrement adaptées pour matérialiser les dépendances entre les éléments de code.* Ils proposent une bonne vision d'ensemble du programme et de la façon dont chaque élément dépend des autres. Ainsi, ils sont de bons candidats pour l'étude de la propagation au sein du programme et de la causalité inhérente à cette propagation. La seconde intuition est le fait que *les données synthétiques sont bien adaptées pour simuler des informations concernant le programme qui seraient sinon difficile à obtenir*, telles que des changements atomiques. Dans le cadre de l'analyse d'impact au changement et de la localisation de fautes, les mutants peuvent être utilisés comme des fautes synthétiques.

1.2 Contributions

Les contributions proposées dans cette thèse sont des réponses directes aux problèmes présentés dans la Section 1.1. La Figure 1 est une représentation simplifiée de la façon dont

les problèmes présentés précédemment et les contribution présentées dans cette sections s'articulent. Comme on peut le voir, le code source peut être utilisé pour trois finalités :

1. produire un *graphe d'appel* ;
2. générer des mutations du code source (appelés mutants),
3. dériver certaines propriétés des logiciels utilisés en entrée d'un générateur de graphes.

Sur base de ces graphes et de ces mutants, quatre contributions sont présentées dans la suite de cette section.

Deux contributions sont liées à l'analyse d'impact au changement, ou en d'autre termes, la détermination d'impacts potentiels (comportements non désirés) liés à un changement (une erreur).

Contribution 1

Un framework permettant l'évaluation des performances pour une technique d'analyse d'impact au changement basé sur les mutants logiciels.

La première contribution résout le premier problème. Ce framework est basé sur des fautes synthétiques obtenues en utilisant la mutation. N'importe quelle technique d'analyse d'impact au changement devra pouvoir être évaluée en utilisant ce framework. Nous avons évalué la prédiction d'impact de quatre types de graphes d'appel. Cette évaluation nous a permis d'étudier la façon dont les erreurs se propagent dans le graphe d'appel sur base d'observations faites sur 16922 mutants créés depuis 10 projets Java libres et en utilisant 5 opérateurs de mutation.

Contribution 2

Une nouvelle technique d'analyse d'impact basée sur les informations apprises de précédents impacts ainsi que sur les graphes d'appel.

La seconde contribution est une nouvelle technique d'analyse d'impact au changement basée sur les graphes d'appel. Les mutants ainsi que les profils d'exécution sont utilisés pour apprendre la cause d'erreurs dans le graphe d'appel, le tout aboutissant en un nouveau type de graphe : le graphe causal. Nous avons évalué notre système en considérant 9 projets Java libres incluant plus de 450000 lignes de code. Nous avons simulé 16682 changements et leur impact réel par le biais de mutants.

Contribution 3

Un nouvel algorithme de localisation de fautes basé sur l'approximation de la causalité et sur les graphes d'appel.

La troisième contribution utilise les graphes ainsi que les mutants pour évaluer leur potentiel dans le cadre de la localisation de fautes ; en d'autres termes, leur capacité à déterminer les causes (les erreurs) à l'origine d'un impact quelconque (le comportement non approprié). Nous proposons d'utiliser les graphes causaux utilisés dans la Contribution 2. Ils sont ici utilisés pour assister et ainsi améliorer les performances de techniques de localisation déjà existantes. Nous avons évalué notre approche sur le jeu de données lié à la localisation de fautes proposé par Steimann et al. incluant un total de 5836 fautes. La troisième contribution répond au second problème.

Contribution 4

Un modèle génératif permettant de créer des graphes logiciels synthétiques.

La grande majorité des travaux présentés dans cette thèse se reposent sur des fautes synthétiques, obtenues par le biais de mutants logiciels. Notre volonté dans cette dernière contribution, est de pousser plus loin l'expérimentation sur les données synthétiques en générant nos propres graphes logiciels synthétiques et ce, en vue de les utiliser pour l'analyse de propagation. Notre dernière contribution est le premier pas vers cet objectif : nous proposons un nouveau modèle génératif permettant la génération de graphes de dépendances synthétiques. Ce modèle est utilisé pour générer des graphes synthétiques similaires aux graphes réels. Nous avons extrait le graphe de dépendances de 50 logiciels Java libres, ce qui représente un total de 23178 noeuds et 108404 arcs que nous avons analysés en vue, d'une part, de générer des graphes similaires à ceux-ci et d'autre part, de comparer leur ressemblance.

Pour résumer, nous proposons dans cette thèse quatre contributions offrant une aide dans le domaine du génie logiciel. Deux sont utilisés pour améliorer l'analyse d'impact au changement, une pour la localisation de fautes et la dernière est un premier pas pour de futurs travaux de recherche.

1.3 Publications

Cette section présente la liste des publications liées aux contributions de cette thèse.

1.3.1 Publiés

[1] Vincenzo Musco, Antonin Carette, Martin Monperrus, and Philippe Preux. A Learning Algorithm for Change Impact Prediction. In *Proceedings of the 5th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering co-located with ICSE*, RAISE '16, pages 8–14, 2016.

[2] Vincenzo Musco, Martin Monperrus, and Philippe Preux. An Experimental Protocol for Analyzing the Accuracy of Software Error Impact Analysis. In *Proceedings of the 10th International Workshop on Automation of Software Test co-located with ICSE*, AST '15, pages 60–64, 2015.

[3] Vincenzo Musco, Martin Monperrus, and Philippe Preux. A Large-scale Study of Call Graph-based Impact Prediction using Mutation Testing. *Software Quality Journal*, 2016. To appear.

[4] Vincenzo Musco, Martin Monperrus, and Philippe Preux. Mutation-based graph inference for fault localization. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation*, October 2016.

La publications [2, 3] concerne la contribution 1, la publication [1] concerne la contribution 2 et la publication [4] concerne la contribution 3.

1.3.2 En cours de soumission

[1] Vincenzo Musco, Martin Monperrus, and Philippe Preux. A Generative Model of Software Dependency Graphs to Better Understand Software Evolution. *Journal of Software : Evolution and Process*, 2016. Minor Revision.

La publication [1] concerne la contribution 4.

1.3.3 A soumettre

[1] Vincenzo Musco, Martin Monperrus, and Philippe Preux. Strogoff : A Recommendation System for Finding Sensitive Method Callers with Weighted Call Graphs.

La publication [1] concerne la contribution 2.

2 Evaluation de techniques d'analyse d'impact

L'analyse d'impact au changement consiste à analyser et à reporter les impacts potentiels relatifs à un changement particulier dans un programme informatique.

L'état de l'art actuel du domaine ne propose pas d'approche automatique permettant de déterminer les performances d'une technique d'analyse d'impact sur un large jeu de données. En effet, un grand nombre de contributions dans le domaine sont peu, voire pas évaluées empiriquement, c'est-à-dire, avec des données réelles. Lorsqu'une évaluation est disponible, elle est en général réalisée sur un nombre très petit de changements. Cette limitation est majoritairement due à l'approche communément empruntée : les changements sont obtenus depuis des différences entre deux versions d'un programme extraites depuis un dépôt de données. Dans une telle approche, un commit dans un dépôt de données ne concerne généralement pas qu'un seul changement, mais une collection de changements. Ainsi, une pré-condition est alors d'extraire d'un commit l'ensemble de changements atomiques associés. Une fois cette condition remplie, déterminer l'impact lié à un changement n'est pas forcément simple. Dans nos travaux, nous proposons une solution permettant d'évaluer les performances d'une technique basée sur un grand nombre de changements sans devoir se tracasser de telles contraintes.

Ainsi, nous proposons un framework d'évaluation pour l'analyse d'impact au changement. L'idée majeure pour obtenir un grand nombre de changements atomiques est d'utiliser le processus de mutation. En effet, en mutant le code source, nous obtenons un grand nombre de copies du programme dans lequel un seul changement atomique a eu lieu. Cette approche est basée sur l'idée de Ali et al. [3] indiquant qu'il n'y a aucune raison de penser qu'un mutant n'est pas comparable à une erreur dans un programme informatique.

Ainsi, pour raisonner sur l'impact d'un changement, nous utilisons la suite de tests du programme. Les tests échouant sont utilisés comme interface permettant de déterminer quels éléments du programme ont été impactés par le changement. En effet, si le test passait avant la mutation et échoue ensuite, il y a une forte probabilité que ce changement de résultat soit provoqué par le changement atomique introduit (la mutation). L'évaluation des performances est réalisée en utilisant les métriques standard : la précision et le rappel. Nous appliquons notre framework sur l'analyse d'impact au changement basée sur les graphes d'appel. Comme il n'y a pas de définition universelle du graphe d'appel, nous avons évalué la performance de quatre types différents de graphes d'appels, chacun ayant sa particularité. Le premier étant obtenu depuis une implémentation externe et les trois autres ont été introduits par nos soins : incluant ou non l'analyse de la hiérarchie d'appel et considérant ou non les dépendances de données.

Nous avons exécuté notre protocole sur 10 programmes Java libres. Sur l'ensemble de ces programmes, nous avons généré un total de 16922 mutants sur base de 5 opérateurs de mutation différents. Ensuite, nous avons comparé la précision et le rappel des différents graphes obtenus. Nos résultats ont montré que la richesse du graphe d'appel augmente le rappel de l'approche. Cependant, à notre grande surprise, le graphe d'appel le plus simple offre le meilleur compromis entre la précision et le rappel déterminé sur base du F-Score.

3 Graphes causaux pour l'analyse d'impact

Nous proposons ici de définir un graphe causal : un nouveau type de graphe obtenu par une phase intermédiaire d'apprentissage réalisée au-dessus d'un graphe d'appel standard (version incluant l'analyse de la hiérarchie de classes). Le but du graphe causal est de proposer une meilleure explication du lien de cause à effet existant entre un élément du code source et un test échouant. Enfin, ce graphe causal est utilisé pour proposer un meilleur filtrage des éléments retournés par un IDE. En effet, les Integrated Development Environment (IDE) sont aujourd'hui considérés comme la référence par les développeurs pour les assister dans leur quotidien. Ils leur permettent d'écrire du code de meilleure qualité en moins de temps et d'efforts. Un IDE possède un nombre important d'outils avancés : refactoring, navigation, lien avec les outils de collaboration tels que les gestionnaires d'erreurs, *etc.* La recherche a permis de proposer un grand nombre d'améliorations des outils de l'IDE au fil du temps, comme par exemple un break-through user-interfaces [6]. Parmi ces éléments proposés, il existe des systèmes de recommandation. L'intuition derrière les systèmes de recommandation est que lorsqu'un développeur manipule un ensemble d'éléments (que ce soit des éléments de code, des erreurs ou autre chose), ils devraient proposer un ordre et/ou un filtrage plus efficace que le simple filtrage aléatoire ou lexicographique.

Un exemple de ces systèmes de recommandation est l'outil intitulé "TrouverAppelants" proposé par l'IDE au développeur. Trouver les appelants, aussi appelé "Trouver les méthodes appelantes" est un outil central pour deux tâches de développement : comprendre comment une méthode est utilisée au sein du programme ainsi que les impacts potentiels en cas de changement de celle-ci.

Dans cette thèse, nous proposons de tirer parti de l'avantage offert par les graphes d'appel d'être de bons candidats pour faire de l'analyse d'impact au changement de telle sorte à filtrer les éléments proposés par l'outil "TrouverAppelants" de l'IDE.

Nous proposons d'apprendre comment les impacts se sont potentiellement propagés basé sur un ensemble de changements et leurs impacts réels. Cette approche s'intitule Strogoff et fonctionne comme telle : un graphe d'appel est construit et ses arcs sont décorés avec des poids allant de 0 à 1, représentant les chances d'être sujet à propager une erreur. Ces poids sont appris en utilisant les résultats obtenus via la mutation du code source. Nous utilisons l'algorithme du plus court chemin entre un changement et un test ayant échoué car nous pensons que les arcs composant ce chemin ont plus de chance de propager un impact.

Les poids des arcs sont mis à jour suivant deux approches différentes : une approche simple où le poids de l'arc prend directement sa valeur maximale si celui-ci fait partie d'un plus court chemin et une approche plus complexe au sein de laquelle le poids de l'arc augmente au fur et à mesure que ledit arc est membre d'un plus court chemin.

Enfin, les méthodes retournées par l'outil "TrouverAppelants" de l'IDE sont filtrées sur base des poids de ces arcs de telle sorte à ne conserver que les éléments réellement sujet à la propagation.

Les performances de Strogoff sont ensuite évaluées sur base du framework d'évaluation (ainsi que le jeu de données) précédemment présenté. Nous considérons 9 programmes Java libres formant un total de 450545 lignes de code. Les graphes d'appel obtenus contiennent un total de 46244 noeuds et 114390 arcs. Nous avons construit un ensemble de 16682 mutants et calculés leurs impacts réels en observant les tests échouant après la mutation. Nous avons ensuite appris les poids sur base de ces changements avec nos deux algorithmes. Ensuite, nous avons simulé des cas d'utilisation dans lesquels nous recherchons des méthodes sensibles. Nos résultats ont montré que Strogoff permet d'obtenir une précision de

65% et un rappel de 76%, correspondant à un F-Score de 58%.

4 Graphes causaux pour la localisation de fautes

Les deux premières contributions ont utilisé les graphes d'appel ainsi que la mutation en vue d'effectuer de l'analyse d'impact au changement sur des programmes Java libres. De plus, nous avons introduit dans la seconde contribution, la notion de graphes causaux, un type spécial de graphe utilisé pour améliorer l'estimation des causes et des conséquences entre les méthodes et les tests. Dans ce chapitre, nous utilisons l'information apprise dans les graphes causaux pour réaliser de la localisation de fautes.

La localisation de fautes est le processus d'identification d'éléments d'un programme qui sont responsables de fautes, c'est-à-dire des éléments qui causent une erreur. Une façon classique d'appréhender le problème de la localisation de fautes est que l'erreur soit reproduite et détectée par un cas de test qui échoue et le but est de prédire la fonction qui contient le code responsable de cette erreur. L'essence du processus de la localisation de fautes est d'essayer de déterminer les relations de causalité entre les éléments du code [4, 9]. En effet, les premiers travaux dans la localisation de fautes étaient basés sur des tranches de programmes (slicing) [2], qui sont des versions plus précises de la relation de causalité la plus évidente : l'erreur doit se trouver quelque part dans le code qui a été exécuté. La localisation de fautes basée sur les spectres représente aussi la causalité jusqu'à un certain point, tout en restant une très forte approximation : les relations de causalité sont seulement représentées par le fait qu'un élément est couvert par des cas de test qui échouent ou réussissent. En effet, la localisation de fautes n'est seulement qu'une approximation de la réelle chaîne de causes et d'effets de la propagation des erreurs qui se passent lors de l'exécution du programme. A notre connaissance, seuls Baah et al. [4] et Shu et al. [9] ont réalisés des étapes majeures en utilisant l'inférence de la causalité pour estimer au mieux les effets des ces causes dans la localisation de fautes.

Nous proposons Vautrin, une approche de la localisation de fautes qui estime la causalité en analysant ce qui se passe entre un cas de test qui échoue et une méthode en utilisant les graphes causaux comme défini dans la Section 3. De façon similaire à Strogoff, Vautrin se base sur l'hypothèse que les mutants et leur profil d'exécution contiennent des informations importantes qui peuvent être utilisées pour estimer la causalité. Basé sur l'information contenue dans le graphe causal et dans un ensemble de tests qui échouent, Vautrin détermine les éléments potentiellement fautifs d'après le graphe. Vautrin ne permet pas d'ordonner les éléments, mais simplement des les filtrer. Quand plusieurs méthodes sont potentiellement fautives, elles sont classées en utilisant les résultats d'une méthode standard de localisation de fautes basée sur les spectres. Dans ce chapitre, nous avont considéré les cinq suivantes : Tarantula, Ochiai, Zoltar, Naish et Steimann.

Afin d'évaluer notre algorithme, nous considérons le jeu de données de Steimann et al. [10] publié à ISSTA'13, composé de 10 programmes Java et de 6 opérateurs de mutation. L'évaluation des performances a été réalisée au moyen du métrique appelé l'effort gâché et celui moins utilisé de la prédiction parfaite qui rapporte le nombre de cas où l'algorithme de localisation de fautes trouve directement la faute réelle. Nous montrons que notre algorithme de localisation de fautes, basé sur les méthodes, surpasse des algorithmes tels qu'Ochiai [1] et Naish [8]. L'amélioration se situe entre 3 et 55% de méthodes en moins à considérer après la localisation de fautes. En utilisant notre technique, sur l'ensemble du jeu de données de Steimann, le nombre de prédictions parfaites est de 2310 sur 5836, ce qui équivaut à 40%, représentant une amélioration de 14% par rapport aux autres travaux.

5 Génération de graphes de dépendances synthétiques

Dans cette thèse, un de nos principaux objectifs est d'utiliser des graphes logiciels et des données synthétiques pour aider les développeurs dans leurs tâches de développement. Dans les chapitres précédents, nous avons présenté comment les graphes d'appel avec des fautes synthétiques (générées via la mutation) peuvent être utilisés pour réaliser de l'analyse d'impact au changement et de la localisation de fautes.

Dans ce dernier chapitre, nous proposons de changer de type de données synthétiques, de passer de la génération de fautes à la génération de graphes. En effet, nous voulons obtenir des graphes synthétiques qui ressemblent à de réels graphes logiciels et qui peuvent être utilisés dans les processus d'apprentissage et de prédiction présentés dans les chapitres précédents.

Nous proposons ici des débuts de travaux relatifs à la génération de tels graphes. Dans ce chapitre, nous considérons les graphes de dépendances logicielles de programmes orienté-objet dans lesquels chaque noeud représente une classe et chaque arc correspond à une dépendance nécessaire au moment de la compilation. Le graphe d'appel (et les graphes causaux) présenté dans les chapitres précédents est aussi un graphe de dépendances, mais au niveau des caractéristiques (méthodes et champs), où seules les méthodes sont considérées. Nous décidons de considérer les graphes de dépendances ayant pour granularité les classes dans le but d'obtenir un vision globale du programme sans les effets de bord des graphes d'appels. Ainsi, nous réduisons les chances d'avoir des graphes trop conséquents comme ceux obtenus en utilisant un processus d'extraction statique ou d'avoir des arcs manquant comme lors d'un processus d'extraction dynamique.

Dans la première partie, nous présentons une étude qui met l'accent sur les topologies communes qui existent dans les graphes de dépendances logicielles. En effet, trouver une topologie commune dans les graphes logiciels est une pré-condition pour proposer un modèle génératif comme nous devons connaître la topologie des données générées. Dans ce but, nous avons extrait les graphes de 50 programmes Java et réalisé des comparaisons statistiques entre chaque paire de programmes, au niveau de la distribution des degrés entrant et sortant.

Dans la seconde partie, nous proposons un modèle génératif de graphes de dépendances logicielles. Comme présenté dans notre intuition de la Section 1.1, nous voulons explorer la capacité des données synthétiques à être utilisées pour améliorer le recherche en ingénierie logicielle. Dans le reste du manuscrit, nous avons considéré les mutants logiciels comme des données synthétiques. Nous proposons ici d'explorer une autre direction où les données synthétiques précédentes (c'est-à-dire les mutants) sont remplacées par un autre type de données synthétiques : les graphes logiciels générés. Nous proposons un modèle génératif basé sur GNC (Growing Network model with Copying), un modèle de génération générique proposé par Krapivski et al. [7]. Nous appelons notre modèle "Generalized Double GNC " (GD-GNC) puisqu'il utilise la primitive d'attachement GNC. Nous l'évaluons avec les données logicielles réelles utilisées dans la première partie, mais aussi par rapport au modèle de Baxter and Frean [5], qui est, selon nos recherches, un des seuls autres modèles prévus pour générer des graphes logiciels similaires aux nôtres. Afin de comparer les deux modèles, nous utilisons des comparaisons similaires à celles utilisées pour la structure commune. Nous explorons aussi d'autres propriétés telles que le diamètre, la longueur moyenne du plus court chemin, la transitivité et la modularité. Ensuite, basé sur les primitives de GD-GNC, nous tentons de fournir une explication spéculative des règles d'évolution qui régissent les programmes. En effet, si ce modèle produit des graphes qui correspondent aux données empiriques, cela signifie que les opérations de génération sont

de bonnes candidates pour décrire les opérations essentielles qui mènent à la structure des graphes logiciels. En d'autres mots, le bon modèle de génération peut inclure les règles d'évolution qui sont derrière la structure des systèmes logiciels.

6 Conclusion et perspectives

Dans cette thèse, nous avons répondu à nos deux problèmes présentés dans la Section 1.1 au moyen de graphes et de données synthétiques obtenues via la mutation. Le premier problème était l'absence d'une méthodologie d'évaluation systématique pour l'analyse d'impact au changement. Le deuxième était que les techniques de localisation de fautes actuelles ne considèrent pas l'ensemble du programme, ignorant comment les éléments dépendent les uns des autres. Ces problèmes ont été évalués durant ma thèse. Dans ce but, quatre contributions ont été proposées. La première contribution était une réponse directe au premier problème. Nous avons proposé un cadre d'évaluation pour les techniques d'analyse d'impact au changement basées sur un grand nombre de changements. L'approche utilise des fautes synthétiques injectées pour contourner la limitation des changements réels : grâce à la mutation, nous nous assurons que le changement inséré est unique. Ainsi, nous pouvons observer les impacts liés à ce changement unique. Basé sur ce système, nous avons mené une étude sur quatre différents types de graphes d'appel pour déterminer leur potentiel pour l'analyse d'impact au changement. Chaque graphe d'appel exprime une caractéristique programmatique de telle façon que nous avons analysé quels sont les éléments les plus responsables de la propagation dans le graphe logiciel. La seconde contribution était Vautrin, un outil destiné à filtrer les sites d'appel retourner par un environnement de développement basé sur les exécutions précédentes du programme. Vautrin dépend du concept de graphe causal : un graphe d'appel sur lesquels les arcs sont décorés avec des poids entre 0 et 1. Ces poids expriment la probabilité d'un arc à propager la faute. Pendant une phase d'apprentissage, des exécutions réelles de tests sont analysées pour saisir les causes et effets des erreurs et des tests dans l'entièreté du graphe. Basé sur cette phase, les poids des arcs du graphe sont mis à jour. Une seuil est utilisé pour désactiver certains arcs et ainsi, filtrer les impacts rapportés. La troisième contribution était Strogoff, une nouvelle technique de localisation de fautes basée sur une phase d'apprentissage similaire à Vautrin, mais où le graphe causal est désormais utilisé pour proposer une solution pour la localisation de fautes. En effet, basé sur le graphe causal, la capacité à déterminer les points d'erreurs basée sur les tests qui échouent nous permet d'obtenir de meilleurs résultats que les techniques existantes selon les principaux métriques d'évaluation. Nous avons utilisé des noeuds reportés comme susceptibles de contenir l'erreur selon le graphe causal en parallèle aux techniques classiques de localisation de fautes basées sur les spectres pour améliorer leur performance. Cette troisième contribution est une réponse à notre second problème puisque utiliser le graphe causal nous permet de raisonner sur le programme dans son entièreté et non d'avoir uniquement une vue limitée d'éléments exécutés. Notre dernière contribution a été le modèle génératif pour les graphes de dépendances logicielles. Ce modèle a été proposé dans un effort pour trouver d'autres données synthétiques à utiliser dans notre recherche. Nous avons d'abord étudié la présence de plusieurs propriétés dans 50 graphes logiciels réels Java. Nous avons trouvé que ces graphes ont une topologie commune au niveau de leur distribution de degrés entrant et sortant. Basé sur ces observations empiriques, nous avons proposé GD-GNC, un modèle génératif basé sur GNC. Ce modèle génératif est capable de générer des graphes logiciels avec des distributions de degrés similaires à des graphes logiciels réels. Cette correspondance a été évaluée avec des distributions réelles et avec

un modèle existant, prévu pour générer le même genre de graphes : celui de Baxter et Frean. Nous avons conclu ce chapitre avec une discussion spéculative des phénomènes qui motivent un telle structure.

Références

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. An Evaluation of Similarity Coefficients for Software Fault Localization. In *Proceedings of the Pacific Rim International Symposium on Dependable Computing*, pages 39–46, 2006.

[2] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault Localization using Execution Slices and Dataflow Tests. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 143–151, October 1995.

[3] Shaimaa Ali, James H. Andrews, Tamilselvi Dhandapani, and Wantao Wang. Evaluating the Accuracy of Fault Localization Techniques. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 76–87, Washington, DC, USA, 2009. IEEE Computer Society.

[4] George K. Baah, Andy Podgurski, and Mary Jean Harrold. Causal Inference for Statistical Fault Localization. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 73–84, 2010.

[5] Gareth. J. Baxter and Marcus R. Frean. Software Graphs and Programmer Awareness. In *ArXiv e-prints*, volume 0802, page 2306, February 2008.

[6] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code Bubbles : Rethinking the User Interface Paradigm of Integrated Development Environments. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 455–464, New York, NY, USA, 2010. ACM.

[7] Pavel L. Krapivsky and Sidney Redner. Network Growth by Copying. *Physical Review E*, 71(3) :036118, March 2005.

[8] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A Model for Spectra-based Software Diagnosis. *ACM Transactions on Software Engineering and Methodology*, 20(3) :11 :1–11 :32, August 2011.

[9] Gang Shu, Boya Sun, Andy Podgurski, and Feng Cao. MFL : Method-Level Fault Localization with Causal Inference. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 124–133, 2013.

[10] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. Threats to the Validity and Value of Empirical Assessments of the Accuracy of Coverage-based Fault Locators. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 314–324, 2013.