Réparation automatique de programmes dans l'IDE

Lom Messan HILLAH et Tewfik ZIADI

Résumé

Ce projet recherche doctoral vise à intégrer au plus près des développeurs, dans le cycle de vie de développement du logiciel, les techniques de réparation automatique de programmes, dont l'adoption est faible dans l'industrie. Ces techniques sont prometteuses dans la facilité qu'elles confèrent à accélérer la détection et la correction automatiques de fautes dans les programmes, contribuant ainsi à l'amélioration de la fiabilité et de la maintenabilité des logiciels.

1 Contexte scientifique

Les technologies du monde moderne sont mises en opération, supervisées et maintenues grâce aux logiciels. De l'industrie automobile à la santé, en passant par le secteur de l'énergie, la banque, finance, assurance ou encore l'industrie spatiale, la demande en logiciel explose. La pandémie de COVID-19 a également engendré une transformation et une accélération numériques durables. Dans ce contexte où le time-to-market se racourcit, les déploiements et correctifs sont fréquents, la pression sur la fiabilité des logiciels s'accroît. Ecrire et maintenir des tests de qualité est un processus pénible et de longue haleine que peu de développeurs et équipes tiennent dans la durée. La génération de tests, la vérification formelle et la preuve automatique de programmes ont fait des progrès considérables ces dernières années [1, 2, 3] mais elles ne sont pas encore massivement diffusées et mises en oeuvre de façon routinière.

Pour soutenir et accélérer le processus de test et correction de fautes afin d'améliorer la fiabilité et la maintenabilité des logiciels, la réparation automatique de programmes (Automated Program Repair - APR) apparaît comme une nouvelle opportunité complémentaire à mettre en oeuvre [4, 5]. Ce domaine de recherche, en pleine expansion, a pour objectif d'intégrer en un seul processus la détection de fautes, leur localisation et leur correction à travers l'élaboration automatique de patchs, leur validation, leur application et la re-validation. Ce processus semi-automatique laisse le développeur maître de la décision de valider et d'intégrer le patch le plus adapté. Chaque étape de ce processus complexe est l'objet d'une thématique de recherche. En plus des

thèmes de recherche évoqués, les enjeux de l'APR concernent la présence et la qualité de tests révélant les fautes, le taux de vrais positifs des outils d'analyse statique, ainsi que l'intégration adéquate de l'outillage dans le cycle de développement du logiciel afin de solliciter efficacement l'attention du développeur au bon moment pour l'application d'un patch. La présente proposition de programme doctoral porte sur ces trois enjeux, dans le but de rendre accessible au plus grand nombre les bénéfices des techniques de réparation automatique de programmes.

Dans le suite de ce document, nous présentons l'essentiel des concepts et techniques afin de se familiariser à ce domaine, puis la problématique que nous proposons de traiter, le travail de thèse proposé et ses pistes de résolution, les moyens d'expérimentation et de validation, enfin les résultats attendus.

2 Concepts et techniques

Une faute, ou bug, est un défaut technique introduit dans le code source d'un logiciel ; une erreur est un état incorrect du logiciel à l'exécution, inobservable, manifestation de la faute qui l'a induite ; une défaillance est un comportement incorrect par rapport au comportement attendu du logiciel [6]. Une défaillance est la manifestation visible de l'occurrence de l'erreur correspondante. Il est donc possible qu'une erreur ne mène pas à une défaillance. Afin qu'une erreur puisse être observée, c'est-à-dire qu'elle provoque une défaillance, elle doit être : i) atteignable, c'est-à-dire pouvoir être localisée dans le code source, ii) infectieuse, c'est-à-dire mener à un état incorrect dans le comportement du logiciel et iii) propagatrice, c'est-à-dire que l'état infecté doit provoquer et révéler un comportement (ou une sortie) incorrect du logiciel.

La gestion d'une faute peut être mise en oeuvre par trois approches complémentaires : i) l'évitement à travers une meilleure conception, ii) la détection à travers le test et le débogage, enfin iii) la tolérance à travers la redondance et l'isolation. Tester consiste à évaluer un logiciel en observant son exécution suite aux stimuli qu'il reçoit ; déboguer consiste à trouver une faute manifestée par une défaillance. Ces deux activités sont difficiles car, d'une part il faut pouvoir produire les données en entrée pouvant déclencher la faute menant à une défaillance, d'autre part étant donnée une défaillance, il faut pouvoir localiser la faute l'ayant déclenchée.

La correction d'un bug requiert de déterminer l'emplacement de la faute puis de déterminer l'emplacement où insérer la correction. L'instruction qui introduit une faute n'est pas forcément celle où il faut appliquer la correction. En effet, l'instruction provoquant la défaillance peut, voire souvent, être différente de celle qui introduit la faute. La détection de la faute peut ne pas repérer précisement l'emplacement de celle-ci. Toute correction peut donc être appliquée soit à l'emplacement de la faute, soit à l'em-

placement ayant provoqué la défaillance, soit à un autre emplacement permettant de contourner la faute entraînant la défaillance.

Les techniques d'APR sont catégorisées selon une approche généraliste ou une approche spécifique [5]. Les techniques généralistes recherchent le maximum de fautes possibles tandis que les approches spécifiques ciblent un type particulier de fautes, par exemple les Null Pointer Exceptions. Une technique peut mettre en oeuvre une méthode de type Generate & Validate [5, 7, 8] ou une méthode de type Semantic-Driven [5, 9]. Chaque méthode peut faire appel à des algorithmes de type Search-based utilisant des heuristiques, ou à des algorithmes de type Brute-force testant chaque possibilité. La génération de patch est définie comme suit. Soit un programme P contenant des problèmes P_{repair}. Une *solution plausible* s_{repair} est recherchée pour chaque P_{repair}. Même si s_{repair} est une solution pour P_{repair}, étant donné que la définition de chaque problème est une approximation, il n'y a aucune garantie que s_{repair} corrige effectivement la faute impliquée.

La méthode Generate & Validate déroule itérativement deux étapes. L'étape de génération construit un ensemble de solutions possibles, S_{repair}, pouvant résoudre ou non P_{repair}. L'étape de validation sélectionne une de ces solutions comme solution plausible pour résoudre P_{repair}. La génération met en oeuvre des opérations de modification, soit unitaire, soit basée sur des templates entièrement construits, soit basée sur des exemples extraits de bases existantes de code. La méthode Semantic-Driven déroule trois étapes. La première rassemble de l'information sur le comportement du programme à partir du résultat des tests ou de l'analyse statique. La deuxième étape construit une représentation formelle du problème P_{repair}. La dernière étape génère, si possible, une solution (donc un patch) pour P_{repair} et retourne le programme réparé. La méthode Semantic-Driven est plus souvent adaptée pour les approches spécifiques ciblant un type particulier de fautes.

3 Problématique

Les outils actuels ont des limitations dans leur mise en oeuvre. Les plus notables concernent la présence et la qualité des tests, l'impact de la taille de l'espace de recherche des patchs sur la performance, ainsi que la validation automatique des patchs générés [10]. Les limitations qui nous concernent sont liées à leur intégration effective dans le cycle de vie du développement logiciel (SDLC) et à l'efficacité de leur mise en oeuvre, i.e., l'impact sur les taux de retour des développeurs et d'acceptation des patchs. Meta (ex Facebook) a déjà relevé que trouver la bonne intégration d'outils de rapports de bug suite à l'exécution des tests et d'outils de suggestions de patch dans le flux de travail des développeurs est déterminant pour le taux de retour de ces derniers [11]. Il

faut rajouter à cette difficulté la diversité des pratiques DevOps, aussi bien au niveau de la configuration des environnements de développement, des méthodologies de revue de code et de test, qu'au niveau des environnements d'intégration continue, de livraison et de déploiement (CI/CD).

Les processus bout-en-bout de détection et de réparation automatique de programmes au sein de workflows CI/CD sont rares et souvent propriétaires. Meta en a déployé une instance en interne [11, 12], qui utilise automatiquement ses propres outils de test et d'analyse statique comme Sapienz [13] et Infer [14]. Le groupe financier Bloomberg en a également déployé une instance, en collaboration avec des universités britanniques [15]. La valeur de tels outils pour ces industriels repose sur les avantages suivants : i) simplicité d'utilisation, ii) efficacité dans la détection et la correction de fautes simples connues et répétitives, iii) passage à l'échelle et rapidité, iv) qualité, lisibilité et compréhension des patchs, v) intégration effective dans le flux de travail des développeurs – pas de rupture ou de dérèglement dans les normes de travail mises en place) et vi) validation par les développeurs. Repairnator [16], seul projet issu de la recherche s'intégrant dans un processus de bout en bout, repose sur des outils APR Open Source et utilise Github ainsi que TravisCI pour détecter des fautes et proposer des patchs à travers des pull requests. Repairnator cible les projets Open Source populaires en Java sur Github ; son fonctionnement est donc particulier de ce point de vue.

Dans une perspective où nous n'avons pas de processus d'intégration continue déterminé à l'avance, de contexte industriel ou de projet déterminé d'avance, comment intégrer de façon harmonieuse dans le flux de travail quotidien de n'importe quel développeur un outillage d'APR, tout en maintenant les avantages mentionnés plus haut et faciliter l'adoption ?

4 Travail de thèse proposé

Nous proposons dans ce projet d'étudier la piste de l'intégration de l'outillage APR dans l'IDE (Integrated Development Environment). Cela tient à plusieurs raisons. La première est que le code et le test d'un logiciel sont principalement produits à partir d'IDE et la concentration du développeur est maximale lorsqu'il travaille sur son code dans l'IDE. Son attention peut être sollicitée dans ce contexte avant qu'il n'ait changé de contexte en passant à une autre tâche. Ensuite, beaucoup d'outils d'analyse statique (e.g., SpotBugs, Sonarlint), de génération de test (e.g., EvoSuite, Ponicode) ou de suggestions automatiques de code (e.g., Github Copilot, Tabnine) sont intégrés dans l'IDE où ils sont efficaces dans l'assistance au développement. Nous pourrions alors tirer parti de l'architecture et des interactions possibles avec ces outils au sein de l'IDE pour renforcer la qualité du travail de l'outillage APR.

Nous étudierons toutes les d'architectures d'intégration possibles, reposant potentiellement sur des composants locaux et distants (dans le Cloud) [17], y compris en s'intégrant à l'ensemble des dépôts de code source d'un même projet (pour faciliter l'analyse interprocédurale), afin de répondre aux critères de passage à l'échelle, rapidité et qualité des patchs. Le travail portera également sur la mise à niveau, l'adaptation et les améliorations algorithmique et architecturale des outils APR qui seront étudiés et sélectionnés, afin qu'ils couvrent un large spectre de frameworks et d'environnements de compilation ou d'exécution dans les langages sélectionnés [1].

4.1 Moyens d'expérimentation et de validation

Nous ciblerons dans ce projet l'intégration dans l'IDE VSCode. Cet IDE a la particularité d'être ouvert, facilement extensible et d'offrir un Language Server Protocol (LSP) [2] qui accélère l'accès au code source de n'importe quel language en vue de réaliser toute sorte d'analyse syntaxique ou sémantique, afin proposer des fonctionalités d'assistance au développement [18]. Nous mènerons une campagne d'invitation aux développeurs de différentes communautés, travaillant sur des projets variés, appartenant à différentes industries, en leur proposant d'utiliser gratuitement le nouvel outillage proposé. Nous définirons et collecterons des métriques sur l'utilisation de cet outillage, ainsi que les feedback des développeurs à travers des enquêtes [19]. Nous analyserons ces métriques et retours afin de construire les profils d'adoption de l'outillage. Cela nous permettra de créer, éventuellement avec des techniques d'intelligence artificielle et de lignes de produits logiciels, un outillage qui saura s'adapter (auto-configuration et suggestion de configuration) au style et aux habitudes de travail de chaque développeur, ainsi qu'à ses bases de code.

Nous analyserons comment l'outillage proposé adhère aux avantages énoncés plus haut et quels taux d'adoption il peut atteindre selon les profils.

4.2 Résultats attendus

Nous souhaitons une intégration sans friction de l'outillage APR dans le flux de travail de la majorité des développeurs au sein de l'IDE. Le travail mené devra montrer, dans ce contexte, quels facteurs ont un impact (positif ou négatif) sur le taux de retour des développeurs dans l'adoption du processus, la validation des patchs proposés et leur intégration dans le code.

1. en particulier Java, très représenté parmi les outils d'APR
2. https ://microsoft.github.io/language-server-protocol/

5 Encadrement

— **Directeur de la thèse** : Dr Tewfik ZIADI, MCF, HDR, chef de l'équipe Modélisation et Vérification (MoVe), LIP6, Sorbonne Université.
— **Encadrant** : Dr Lom Messan HILLAH, MCF, Université Paris Nanterre et LIP6 (MoVe)/Sorbonne Université.

5.1 Collaborations envisagées

Ce travail pourra être mené en collaboration avec les chercheurs travaillant sur l'outil Repairnator et Fixie de Bloomberg, car ils ont acquis une expérience soit de projets Open Source, soit de projets industriels, dont les leçons apprises nous seront utiles pour donner une orientation stratégique au présent projet doctoral.

Références

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

[2] M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. An industrial evaluation of unit test generation : Finding real faults in a financial application. In *2017 IEEE/ACM 39th International Conference on Software Engineering : Software Engineering in Practice Track (ICSE-SEIP)*, pages 263–272. IEEE, 2017.

[3] Nathan Chong, Byron Cook, Jonathan Eidelman, Konstantinos Kallas, Kareem Khazem, Felipe R. Monteiro, Daniel Schwartz-Narbonne, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. Code-level model checking in the software development workflow at Amazon Web Services. *Software : Practice and Experience*, 51(4) :772–797, April 2021.

[4] Martin Monperrus. Automatic software repair : a bibliography. *ACM Computing Surveys (CSUR)*, 51(1) :1–24, 2018.

[5] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair : A survey. *IEEE Transactions on Software Engineering*, 45(1) :34–67, 2017.

[6] ISO/IEC JTC1/SC 7, Sofware and systems engineering committee. ISO/IEC/IEEE 24765 :2017(en), Systems and software engineering — Vocabulary. International Standard, ISO, 2017.

[7] Matias Martinez and Martin Monperrus. Astor : A program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 441–444, 2016.

[8] Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, volume 1, pages 213–224. IEEE, 2016.

[9] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix : Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE, 2013.

[10] He Ye, Matias Martinez, and Martin Monperrus. Automated patch assessment for program repair at scale. *Empirical Software Engineering*, 26(2) :20, February 2021.

[11] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. Sapfix : Automated end-to-end repair at scale. In *2019 IEEE/ACM 41st International Conference on Software Engineering : Software Engineering in Practice (ICSE-SEIP)*, pages 269–278. IEEE, 2019.

[12] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix : Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA) :1–27, 2019. Publisher : ACM New York, NY, USA.

[13] Ke Mao, Mark Harman, and Yue Jia. Sapienz : Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 94–105, 2016.

[14] Infer. https://fbinfer.com/.

[15] Serkan Kirbas, Etienne Windels, Olayori McBello, Kevin Kells, Matthew Pagano, Rafal Szalanski, Vesna Nowack, Emily Rowan Winter, Steve Counsell, David Bowes, et al. On the introduction of automatic program repair in bloomberg. *IEEE Software*, 38(4) :43–51, 2021.

[16] Simon Urli, Zhongxing Yu, Lionel Seinturier, and Martin Monperrus. How to design a program repair bot ? insights from the repairnator project. In *2018 IEEE/ACM 40th International Conference on Software Engineering : Software Engineering in Practice Track (ICSE-SEIP)*, pages 95–104. IEEE, 2018.

[17] Linghui Luo, Martin Schäf, Daniel Sanchez, and Eric Bodden. IDE support for cloud-based static analyses. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1178–1189, Athens Greece, August 2021. ACM.

[18] Nadeeshaan Gunasinghe and Nipuna Marcus. *Language Server Protocol and Implementation*. Apress, Berkeley, CA, 1^{st} edition, 2022.

[19] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Communications of the ACM*, 61(4) :58–66, 2018.