Permettre la programmation sans bugs
Défi GDR GPL

Gabriel Scherer, INRIA Saclay

2 décembre 2019

Résumé

Aujourd'hui la majorité des langages utilisés en pratique pour écrire du logiciel ne permettent que de décrire l'*implémentation* du logiciel désiré. Pour réduire drastiquement le nombre de bugs logiciels, il faudrait permettre l'utilisation d'un langage dit *vérifiant*, c'est-à-dire un langage qui permet aussi d'écrire la *spécification* du logiciel et de prouver que l'implémentation la respecte – la preuve étant vérifiée par l'environnement de programmation.

Ce projet de recherche au long cours a déjà donné des succès remarquables, avec des compilateurs, des systèmes d'exploitation, des serveurs webs vérifiés ; mais ces succès restent des prototypes de recherche dont l'écriture est réservée à des super-experts.

Notre défi est de produire des langages vérifiants utilisables en pratique. Cela demande de nombreuses améliorations, en particulier de meilleurs langages vérifiants, de meilleurs outils de preuve/vérification, et des utilisateurs et utilisatrices mieux formé-e-s.

Introduction

Un *bug* logiciel, c'est un écart de comportement entre ce que fait le programme et ce qu'on aurait souhaité qu'il fasse. Ce n'est pas une erreur de l'ordinateur, c'est une erreur humaine : la machine fait toujours ce qu'on lui a demandé, mais on s'est trompé, on n'a pas demandé ce qu'on voulait vraiment.

De même qu'un système intégrant des humains et des machines contiendra toujours des failles de sécurité, il n'est pas possible de promettre l'absence totale de bugs. Le *défi* que nous proposons est de permettre aux auteurs de logiciel d'exprimer précisément *ce qu'ils veulent*, c'est-à-dire une *spécification* pour leur programme, et d'avoir des outils pour vérifier que ce qu'ils ont demandé, leur *implémentation*, respecte cette volonté exprimée.

On appellera *langage vérifiant* un langage de programmation qui permet d'écrire des spécifications en plus des implémentations, et de vérifier statiquement qu'elles sont respectées. Le défi est de produire des langages vérifiants utilisables en pratique. Même dans un monde idéal où ce défi est résolu, il restera toujours des fissures où des bugs peuvent se glisser, qu'il est important de reconnaître et de nuancer/relativiser.

— On ne connaît pas forcément la spécification du programme que l'on veut écrire. Quelle serait la spécification d'un économiseur d'écran, "afficher de jolies images" ? Mais même dans ce cas on peut donner une *spécification partielle*, qui parle de certain comportements du programme : l'économiseur d'écran ne doit pas s'arrêter avant une action de l'utilisateur ou la mise en veille de la machine. Par ailleurs, un logiciel dont nous n'avons pas de bonne spécification est la partie émergée d'un iceberg logiciel, reposant sur une grande quantité de logiciel système et de bibliothèques pour lesquelles on sait écrire des spécifications, que l'on voudrait vérifier.

— On peut encore se tromper en écrivant la spécification ! Cependant, dans la majorité des cas les spécifications sont des objets beaucoup plus simples et compacts que le code source ; elles sont plus faciles à relire et comprendre, exposent moins de surface aux bugs. (Une spécification qui se contente de répéter l'implémentation n'apporte rien, et se rapproche du cas où on ne sait pas spécifier.) On peut aussi compter sur des méthodes de tests de

spécifications, plus efficaces que les tests d'implémentation car elles opèrent sur un objet plus simple.

Avantages indirects

Écrire des spécifications ne sert pas qu'à éliminer des bugs. Nous prétendons que le fait d'encourager les programmeu-r-se-s à écrire des spécifications en même temps que leur implémentation augmente significativement la qualité logicielle.

1. Se demander systématiquement la spécification d'une nouvelle fonction ou méthode, se forcer à réfléchir plus en écrivant du code, aide à écrire le code correspondant — du Test-Driven-Development (TDD) amplifié.
2. Réfléchir à la spécification aide à concevoir l'interface (API) d'une bibliothèque. Dans tel cas particulier, quelle valeur par défaut devrait utiliser la fonction ? La valeur qui permet la spécification la plus simple.
3. Les spécifications constituent une forme précieuse de documentation. En particulier, si elles sont vérifiées automatiquement, on a la garantie qu'elles sont à jour — contrairement aux commentaires libres.
4. La présence de spécifications pourra aider grandement les outils de transformation automatique de programme (refactoring, génération de code à partir d'exemples, mise à jour automatisée, etc.) : on réduit l'espace des transformations possibles en se restreignant à celles qui valident la spécification.

Pour l'instant nous n'avons que relativement peu d'expériences concrètes sur l'impact des langages vérifiants en terme de génie logiciel, mais les trois premiers points peuvent être constatés (nous en avons fait personnellement l'expérience) en utilisant des outils de génération de tests aléatoires, en particulier le *test basé sur les propriétés*, qui encourage à écrire des spécifications partielles.

De meilleurs langages

Il existe aujourd'hui très peu de langages vérifiants, en tout cas parmi les langages de programmation généralistes. Comme outils et prototypes de recherche, on peut citer Why3 et Dafny, F* et Liquid Haskell, Coq et Idris. SPARK/Ada est peut-être le seul exemple utilisé en production dans l'industrie.

L'approche dominante est l'utilisation de systèmes de type et d'analyses statiques automatiques, qui vont dans cette direction mais donnent des spécifications très partielles ("renvoie un entier", "attend un tableau de taille N", "ne plante pas"). Les mécanismes de *programmation par contrat* méritent une mention honorable, car ils encouragent les programmeurs à écrire des spécifications. Mais ils sont vérifiés dynamiquement : ils n'apportent pas la confiance d'une vérification statique, et leur coût de calcul limite la richesse des contrats employés en pratique.

Pour relever ce défi, il faudra concevoir de meilleurs langages vérifiants, et ajouter des bons langages de spécification (et outils de vérification) aux langages existants — les propositions existantes comprennent JML pour Java, ACSL pour le C, le projet VOCAL pour OCaml. Un langage de spécification n'est pas la même chose qu'un langage de contrats (habituellement écrits dans le langage d'implémentation) ; même si on peut vouloir le tester dynamiquement, il doit pouvoir exprimer des propriétés non calculables, doit pouvoir être évalué statiquement, et demande souvent des mécanismes de réutilisation et modularité un peu différents du langage d'implémentation.

La conception d'un langage de spécification est aussi intimement liée à la conception du langage d'implémentation, ce ne sont pas deux objets totalement séparés. Par exemple, notre expérience collective en vérification de programme souligne l'importance de la notion d'*état fantôme*, qui consiste à entrelacer dans les instructions du programme des calculs "fantômes" qui ne sont pas effectués dynamiquement, servent uniquement à la vérification, en maintenant une relation précise entre l'évolution des données (réelles) du programme et la spécification de haut niveau. Par exemple, à un tableau représentant une queue FIFO on pourra associer un ensemble "fantôme" de ses

éléments, maintenue en parallèle et utilisée dans les spécifications. Cette construction est spécifique aux langages vérifiants.

En plus de questions de design et d'utilisabilité, cette question touche des sujets théoriques de *logique* : quelle est la bonne logique pour formuler des énoncés mathématiques concernants des programmes contenant des exceptions, des effets de bord, de la concurrence par passage de messages ou mémoire partagée, des systèmes distribués ? Le sujet est très ancien, mais par exemple, la *logique de séparation*, apparue au début du siècle et toujours un sujet de recherche actif, a révolutionné la façon de parler de l'état modifiable dans des spécifications.

De meilleurs outils de preuve

Un langage vérifiant demande des outils de preuve, qui peuvent vérifier que les implémentations respectent leur spécification. Le problème de construire de bons vérificateurs est intimement lié aux choix du langage de spécification et du langage d'implémentation, mais il touche aussi à des problèmes fondamentaux de recherche de preuve, démonstration automatique et assistants de preuve.

Malgré les progrès rapides et constants de outils de preuve automatique, l'expérience de notre communauté est qu'espérer une preuve totalement automatisée n'est pas une bonne approche : dans les cas délicats les outils automatiques deviennent lents et fragiles (la vérification ne passe plus après des changement mineurs du programme, réussit ou échoue selon les choix non-déterministes de l'outil de preuve).

En plus de pousser pour une meilleure vérification automatique des obligations de preuve issues de la vérification de programmes, tous les systèmes existants travaillent donc sur des façons pour l'utilisateur de guider la preuve que l'implémentation respecte la spécification. De nombreuses pistes sont explorées ; par exemple, faut-il essayer de pousser le guidage de la vérification au maximum comme une forme de programmation (fonctions-lemmes, état fantôme), ou au contraire utiliser un langage de preuve généraliste, permettant de travailler sur n'importe quel énoncé mathématique ?

Les outils de vérification posent aussi de nouvelles questions de conception logicielle. Quelle est la meilleure façon de faire évoluer une spécification en maintenant à jour les arguments de preuve ? Quels sont les approches de preuve qui donnent les arguments les plus robustes aux évolutions du programme ?

De meilleur-e-s programmeu-r-se-s

Indépendamment de tous les efforts passés et à venir de notre communauté pour rendre les langages *vérifiants* plus faciles à utiliser, l'écriture de spécification reste un art différent de l'écriture d'implémentation, qui appelle une formation spécifique.

Pour relever ce défi, tous les cursus de master d'informatique devraient contenir un cours obligatoire de vérification de programmes, enseignant l'usage d'un langage vérifiant.

Remerciements Nous remercions Arthur Charguéraud et Jean-Christophe Filliâtre pour leurs commentaires sur ce document.