Vérification déductive de programmes Java avec l'outil KeY

Simon Robillard

27 septembre 2022

Les bugs, c'est grave ?

Les bugs, c'est grave ?

Les défauts dans des systèmes informatiques peuvent causer un perte de...

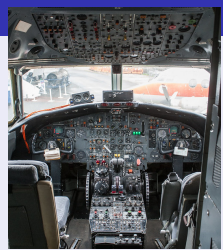
temps...	argent ...	données privées ...	ou pire ...
			
USS Yorktown	Pentium FDIV	Heartbleed	Lime scooters
arrêté pendant 3 heures	$475 millions	TLS compromis	blocage des freins à haute vitesse

Approche traditionnelle pour garantir la qualité d'un système logiciel : les tests

- conceptuellement faciles à mettre en œuvre
- utiles à différentes étapes de la vie du logiciel
  - développement (tests unitaires, tests d'intégration, *test-driven development*)
  - livraison du logiciel (tests de validation)
  - mises à jour (tests de non-régression)

Exemple : logiciels critiques en avionique

- normes ED-12C et DO-178C (*Software considerations in airborne systems and equipment certification*)
- classifie les systèmes selon la gravité des problèmes qu'un bug pourrait entraîner
- spécifie différents niveaux de tests suivant les catégories



Peut-on tout tester ?

"Tester des programmes peut être un moyen très efficace de révéler des bugs, mais est irrémédiablement inadapté pour en démontrer l'absence."

– Edsger W. Dijkstra

Issu de *The Humble Programmer*, discours prononcé lors de la réception du prix Turing en 1972

Tests et exhaustivité

Tester tous les inputs de la fonction `abs(n: int)` ?

Tests et exhaustivité

Tester tous les inputs de la fonction `abs(n: int)` ?

- si `int` est un entier 64-bit

$$2^{64} = 18\ 446\ 744\ 073\ 709\ 551\ 616 \text{ possibilités*}$$

*environ 100 fois le nombre de secondes écoulées depuis le Big Bang

- avec des entiers multiprécision (e.g., Python), nombre infini de possibilités

Tests et exhaustivité

Tester tous les inputs de la fonction `abs(n: int)` ?

- si `int` est un entier 64-bit

$$2^{64} = 18\ 446\ 744\ 073\ 709\ 551\ 616 \text{ possibilités*}$$

*environ 100 fois le nombre de secondes écoulées depuis le Big Bang

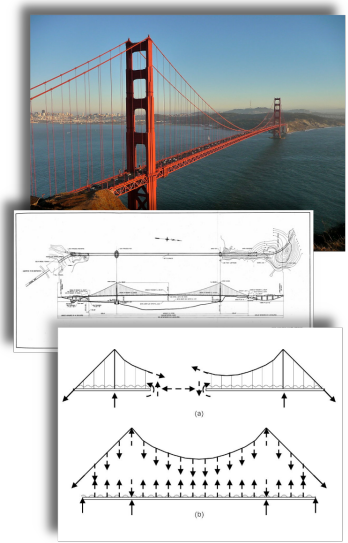
- avec des entiers multiprécision (e.g., Python), nombre infini de possibilités

Couverture de code

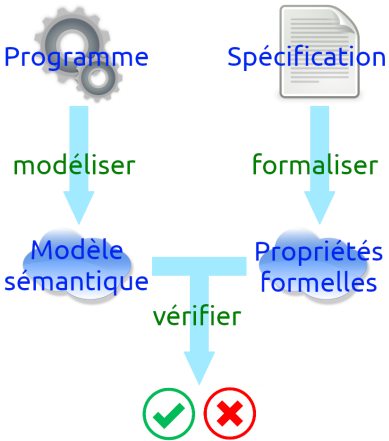
- une autre façon de définir l'exhaustivité
- différents niveaux possibles : exécuter toutes les fonctions/instructions/branches/conditions du programme
- exemple : critère MC/DC pour les logiciels critiques dans la norme DO-178
- impossible de déterminer la couverture sans avoir écrit le code

Les méthodes formelles

- pour remédier à ces problèmes
  - représentations abstraites
  - techniques de raisonnement mathématiques
- c'est l'approche standard pour les ingénieurs dans tous les autres domaines !
- en informatique, on appelle *méthodes formelles* l'ensemble de ces représentations et techniques



Abstraction



- la vérification travaille sur une représentation mathématique du programme
- permet d'utiliser des techniques mathématiques pour analyser toutes les entrées (même un nombre ∞)
- pas besoin d'exécuter le programme/système
- demande une abstraction fiable

Tour d'horizon (non-exhaustif) des méthodes formelles

il existe diverses formes de méthodes formelles

- différents modèles pour différents systèmes
  - modèle abstrait (haut-niveau) ou plus détaillé (bas-niveau) ?
  - code impératif ou déclaratif ?
  - programme séquentiel ou système distribué ?
  - ...
- différents types de propriétés
  - propriétés fonctionnelles, propriétés temporelles, propriétés de sécurité...
  - spécificités du paradigme de programmation (ex : concurrence, calcul sur les flottants)
  - spécificités du domaine d'application

Tour d'horizon (non-exhaustif) des méthodes formelles

Typage

- **idée** : le typage des fonctions peut être suffisament précis pour exprimer exactement ce que le code doit faire. Le compilateur n'accepte que les fonctions qui satisfont leur type.
- **avantages** : permet de spécifier très finement les fonctions. Le langage de programmation *est* le langage de spécification.
- **limites** : le compilateur a besoin d'indications de typage fournies par le développeur (= une preuve). Demande beaucoup de temps et de connaissances en logique
- **outils** : Coq, Agda, PVS...

Tour d'horizon (non-exhaustif) des méthodes formelles

Model-checking

- **idée** : représenter les états/transitions d'un système sous forme de graphe et utiliser un algorithme de de parcours pour vérifier les propriétés du graphe.
- **avantages** : permet de spécificier le comportement temporel des systèmes. Le processus de vérification est automatique.
- **limites** : le graphe est immense, on ne l'explore souvent qu'à une profondeur donnée (*bounded model checking*). Le niveau d'abstraction est très haut.
- **outils** : Spin, TLA+, UPPAAL, ...

Tour d'horizon (non-exhaustif) des méthodes formelles

Raffinement

- **idée** : partir d'un modèle abstrait, et le transformer plusieurs fois jusqu'à obtenir une version exécutable
- **avantages** : procéder par étapes successives permet de limiter la difficulté. La méthode s'intègre à un processus de développement.
- **limites** : chaque transformation doit être prouvée
- **outils** : méthode B

Vérification déductive

Dans cette présentation : *vérification déductive* de *propriétés fonctionnelles*

- système = programme (en partant du code source, ici en Java)
- propriétés à vérifier = relations entre les (états d') entrées du programme et ses (états de) sorties

$P(\sigma)$ — pré-condition

programme à vérifier

$Q(\sigma')$ — post-condition

Section 2

Le projet KeY

- projet initié en 1998 au Karlsruhe Institute of Technology (Allemagne)
- aujourd'hui développé avec TU Darmstadt (Allemagne) et Chalmers Universiy of Technology (Suède)
- principalement destiné à la vérification fonctionnelle de code Java
- logiciels (libres) et littérature disponible
  - prouveur
  - symbolic execution debugger
  - KeY book

Spécifier des programmes Java

Java Modeling Language (JML)

- commentaires Java avec une syntaxe spéciale
- permet de définir des *contrats* pour chaque méthode
  - pré-condition(s) sur les arguments passés en entrée
  - post-condition(s) sur les valeurs retournées ou modifiée en place
- possibilité de définir plusieurs contrats par méthode, pour différents cas d'utilisation, y compris les cas où la méthode renvoie une exception

Caractéristiques de JML

- les pré/post-conditions sont des formules de logique du 1er ordre
  - quantificateur `\forall`
  - quantificateur `\exists`
- le mot clé `\result` désigne la valeur retournée
- dans les post-conditions, le qualificateur `\old` permet de désigner la valeur d'une variable avant application de la méthode (utile pour les méthodes qui modifient des objets en place)
- quelques compréhensions numériques (`\sum`, `\product`, `\min`, `\max`...)

Démo spécification

La logique dynamique

une logique avec les opérateur booléens classiques + deux opérateurs spéciaux qui permettent de mêler du code dans la logique

- modalité *box*

$$[p]\varphi$$

"le programme p termine dans un état qui satisfait la formule φ"

- modalité *diamond*

$$\langle p \rangle \varphi$$

"si le programme p termine, alors il le fait dans un état qui satisfait la formule φ"

Le processus de vérification d'une formule DL

Soit une méthode dont le code est le programme p, et un contrat avec

- des pré-conditions $P_1, \ldots, P_n$
- des post-conditions $Q_1, \ldots, Q_m$

Il faut alors prouver la formule

$$P_1 \wedge \cdots \wedge P_n \implies \langle p \rangle Q_1 \wedge \cdots \wedge Q_m$$

pour établir que la méthode satisfait le contrat (remplacer la modalité par *box* si on veut aussi vérifier la terminaison)

Inférences pour la logique dynamique

La logique dynamique est une extension de la logique du premier ordre.
On peut étendre le calcul des séquents pour prouver la validité des formules.

$$\frac{\vdash \varphi}{\vdash \langle\rangle\varphi} \qquad \frac{\vdash \varphi}{\vdash []\varphi} \qquad \frac{\vdash [p]\varphi \qquad \vdash \langle p\rangle\top}{\vdash \langle p\rangle\varphi}$$

$$\frac{\vdash \{v := e\}\langle p\rangle\varphi}{\vdash \langle v = e; p\rangle\varphi} \qquad \frac{\vdash \varphi[v \mapsto e]}{\vdash \{v := e\}\varphi} \qquad \frac{c \vdash \langle p_1\rangle\varphi \qquad \neg c \vdash \langle p_2\rangle\varphi}{\vdash \langle \texttt{if}\,(c)\,\{p_1\}\,\texttt{else}\,\{p_2\}\rangle\varphi}$$

Prouver une formule de logique dynamique

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x >= 0 \vdash x >= 0}}{x >= 0 \vdash \{r := x\}\, r >= 0}}{x >= 0 \vdash \{r := x\}\langle\rangle\, r >= 0}}{x >= 0 \vdash \langle r = x;\rangle\, r >= 0} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{x < 0 \vdash x <= 0}}{\neg x >= 0 \vdash -x >= 0}}{\neg x >= 0 \vdash \{r := -x\}\, r >= 0}}{\neg x >= 0 \vdash \{r := -x\}\langle\rangle\, r >= 0}}{\neg x >= 0 \vdash \langle r = -x;\rangle\, r >= 0}$$

$$\dfrac{\vdash \langle \texttt{if (x >= 0) \{ r = x; \} else \{ r = -x; \}} \rangle\, r >= 0}{\vdash \top \implies \langle \texttt{if (x >= 0) \{ r = x; \} else \{ r = -x; \}} \rangle\, r >= 0}$$

Prouver des programmes avec des boucles

- en général les modalités peuvent être simplifiées automatiquement, pour ne laisser qu'une formule de logique du 1er ordre que le prouveur doit résoudre
- ce n'est pas le cas si le programme contient des boucles
- l'utilisateur doit alors donner pour chaque boucle un *invariant*, c.-à-d. une formule qui
  - soit vraie au début de la première itération
  - soit préservée par l'exécution de la boucle
  - donne assez d'information prouver les post-conditions

Stratégie pour trouver des invariants

1. utiliser des motifs récurrents (en particulier pour les boucles sur des indices de tableau)
2. partir d'une propriété et la généraliser jusqu'à obtenir un propriété préservée par la boucle
   - pré-condition (exécuter symboliquement la boucle sur cette propriété)
   - post-condition (calculer la condition initiale nécessaire pour valider la post-condition)

Preuves de terminaison

- les exemples précédents terminaients tous trivialement : pas de différence *box* et *diamond*
- cas non-triviaux
  - boucles
  - fonctions récursives
- il faut prouver la terminaison à l'aide d'un *témoin*
  - expression numérique dont la valeur décroit strictement à chaque itération/récursion
  - positive
  - annotation ajoutée au programme

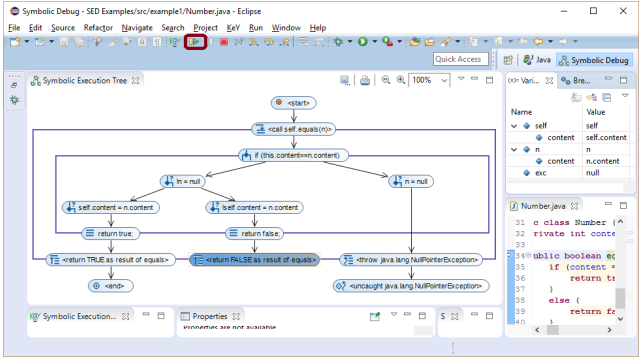
Démo terminaison

Spécification et mémoire

- *effets de bord* = interaction d'une méthode avec la mémoire en dehors de ses arguments/valeur de retour
  - en lecture : par ex., la méthode lit une variable globale
  - en écriture : par ex., la méthode trie un tableau en place
- problème pour la vérification : il faut expliciter ce que la méthode *ne fait pas* !
  - modificateur `pure` si la méthode n'a aucun effet de bord
  - sinon, clause `modifies` ou `assignable` pour dénoter les objets modifiés par la méthode

Symbolic Execution Debugger

- résoudre une formule de DL = exécuter symboliquement le programme en partant d'un état initial décrit par une formule
- SED = debugger capable d'exécuter symboliquement un prg. sans valeurs initiales



Section 3

Étude de cas : le bug Timsort

- le chercheur Stijn de Gouw avait déjà utilisé KeY pour vérifier les algorithmes de tri *radix sort* et *counting sort*
- en 2015, il se tourne vers l'implémentation OpenJDK de *Timsort*
  - code complexe et optimisé
  - utilisé dans des milliers d'applications

TimSort

- développé en 2002 par Tim Peters pour Python
- utilisé dans les librairies
  - Python
  - Java SE
  - Android
  - GNU Octave
  - V8 (moteur JavaScript)
  - Rust
  - Swift

Le fonctionnement de l'algorithme

- de gauche à droite, recherche des sous-segments déjà triés ("monotonies")
- si le sous-segment trouvé est trop court, il peut être allongé via un tri par insertion

$$\{\ 4,\ 3,\ 12,\ 7,\ 9,\ 11,\ 32,\ 9,\ 11,\ 19,\ 23,\ 25,\ 37,\ 44\ \}$$

- l'algorithme retient la longueur des monotonies trouvées dans une pile `runLen`
  - ici $\texttt{runLen} = \{1, 2, 4, 7\}$
- `runLen` doit satisfaire les propriété suivante
  - $\forall i,\ \texttt{runLen}[i] > \texttt{runLen}[i-1]$
  - $\forall i,\ \texttt{runLen}[i] > \texttt{runLen}[i-1] + \texttt{runLen}[i-2]$
- à chaque ajout d'une nouvelle monotonie à `runLen`, la méthode `mergeCollapse()` vérifie que l'invariant est respecté, sinon elle lance un tri fusion entre deux monotonies pour le rétablir

Le bug

- pour des raisons de performance, la taille allouée à `runLen` est seulement aussi grande que nécessaire
- l'invariant permet de déduire quelle taille allouer à `runLen` en fonction du nombre d'éléments à trier

Le bug

- pour des raisons de performance, la taille allouée à `runLen` est seulement aussi grande que nécessaire
- l'invariant permet de déduire quelle taille allouer à `runLen` en fonction du nombre d'éléments à trier
- problème :
  1. la méthode `mergeCollapse()` est buguée et ne maintient pas toujours l'invariant
  2. la pile `runLen` contient alors plus d'éléments que prévu
  3. l'exécution termine avec une `ArrayOutOfBoundsException`

Vérification de TimSort

KeY a été utilisé pour la (tentative de) vérification

- sert normalement à prouver qu'un programme est correct, pas à trouver des bugs !
- la preuve semi-interactive permet néanmoins de comprendre le programme
- branche non fermée = problème potentiel
- Stijn de Gouw a utilisé cette information pour
  1. comprendre le bug
  2. produire des valeurs qui font crasher le programme
  3. proposer un fix
- la version reparée a été prouvée correcte avec KeY

Section 4

Conclusions : leçons à tirer

La vérification dans la vie d'un développeur

- Keep It Simple, Stupid
  - le code complexe cache très probablement des bugs
  - si vous ne pensez pas pouvoir prouver votre code, il est peut-être trop complexe
  - éviter les effets de bord, privilégiez les méthodes courtes...
- mieux que les commentaires, écrivez des assertions
  - en particulier les pré-requis des méthodes, mais aussi certains invariants ou post-conditions non-triviales
  - permettent de repérer les problèmes à la source plutôt que les conséquences
  - seront mis à jour si le code change
  - désactivables pour le code mis en production