Vérification déductive de programmes Java avec l'outil KeY

Pourquoi vérifier des programmes ?

Les bugs dans les programmes peuvent provoquer :

- Une perte de temps
- Une perte d'argent
- Des fuites de données
- Des accidents mortels

Comment vérifier des programmes ?

Il y a 2 manières de vérifier l'existance de bugs dans un programme :

- En effectuant des tests (unitaires, d'intégration, de développement, de validation et de non régression)

  > Efficace pour révéler des bugs mais inadapté pour en montrer l'absence.
  > Par exemple si on souhaite montrer que la fonction valeur absolue `abs(n: int)` ne présente aucun bug, il faut tester tous les cas d'entier possible.
  > Si les entiers sont codés sur 64-bit il y a donc 2^{64} possibilités.

- En utilisant des méthodes formelles de détections de bugs

  > La vérification est réalisée sur le modèle sémantique d'un programme ainsi que sur les propriétés formelles liées à sa spécification.
  > Cette vérification s'opère sur une représentation mathématique du programme ce qui permet d'utiliser des techniques mathématiques pour analyser toutes les entrées (même un nombre infini) et ce, sans avoir à exécuter le programme.

  Il existe diverses formes de méthodes formelles :

  - Différents modèles pour différents systèmes
  - Différents types de propriétés

  Dans cette présentation on s'intéresse à la vérification fonctionnelle :

  - Le système désigne le programme (le code source Java)
  - Les propriétés à vérifier sont : les relations entre les états d'entrées du programme et ses états de sorties (on définit donc une pré-condition sur les données en entrée et une post condition sur les données en sortie)

Le projet KeY

Informations à connaitre sur le projet KeY :

- Initié en 1998 en Allemagne

- Aujourd'hui développé en Allemagne et en Suède
- Destiné à la vérification fonctionnelle de code Java
- Il s'agit d'un logiciel libre

Utilisation du logiciel KeY pour vérifier ses programmes Java :

- Utilisation du `Java Modeling Language (JML)`

  > Il s'agit de commentaires Java ayant une syntaxe particulière.
  >
  > Ça permet de définir des `contrats` sur chaque méthode (pré-condition(s) sur les inputs / post-condition(s) sur les outputs).
  >
  > Il est possible de définir plusieurs contrats sur une même méthode, pour différents cas d'utilisation de la méthode même dans le cas où la méthode renvoie une exception.

  - Caractéristiques de JML
    - Les pré/post-conditions sont des formules de logique du premier ordre (présence de quantificateur)
    - Le mot clef `\result` désigne la valeur retournée
    - Dans les post-conditions, le qualificateur `\old` désigne 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` ...

- Utilisation de la `logique dynamique`

  > Il s'agit d'une logique avec des opérateurs booléens classiques et deux opérateurs spéciaux permettant de mêler du code dans la logique.

  - modalité `box` :

    $[p]\phi$ signifie "le programme p termine dans un état qui satisfait la formule ϕ"

  - modalité `diamond` :

    $< p > \phi$ signifie "si le programme p termine, alors il le fait dans un état qui satisfait la formule ϕ"

- Processus de vérification d'une formule de logique dynamique

  > On suppose une méthode dont le code est p et un contrat avec les pré-conditons $P_1, \ldots, P_n$ et post-conditions $Q_1, \ldots, Q_m$.
  >
  > Il faut alors prouver la formule : $P_1 \wedge \ldots \wedge P_n \Rightarrow < p > Q_1 \wedge \ldots \wedge Q_n$ pour établir que la méthode satisfait bien le contrat.

  exemple avec $p \equiv$ `if (x >= 0) then r := x else r := -x` :

  $\top \Rightarrow$ `<p>r` $\geqslant O$

  $\iff$

  $\bot \vee$ `<if (x >= 0) then r := x else r := -x>r` $\geqslant O$

  $\iff$

  `<if (x >= 0) then r := x else r := -x>r` $\geqslant O$ `On évalue que l'expression de droite car l'expression de gauche est fausse`

  $\iff$

  $x \geqslant 0 \Rightarrow$ `<r := x>r` $\geqslant O \ \wedge \ \neg(x \geqslant 0) \Rightarrow$ `<r := -x>r` $\geqslant O$

$$\iff$$

$$x \geqslant 0 \Rightarrow x \geqslant O \ \wedge \neg(x \geqslant 0) \Rightarrow -x \geqslant O$$

$$\iff$$

$$\top$$

- Prouver des programmes avec des boucles

  Si le programme contient des boucles, l'utilisateur doit donner pour chaque boucle un `invariant`, c-à-d une formule qui :

  - est vraie pour la première itération
  - soit toujours vraie pendant les itérations suivantes
  - donne assez d'information pour prouver les post-conditions

  Stratégie pour trouver des invariants :

  - utiliser des motifs récurrents (notamment pour les boucles sur des indices de tableau)
  - partir d'une propriété et la généraliser jusqu'à obtenir une propriété préservée par la boucle

- Preuves de terminaison

  Il y a des cas de terminaison non triviaux :

  - les boucles
  - les fonctions récursives

  On prouve la terminaison à l'aide d'un `témoin` :

  - expression numérique positive dont la valeur décroit strictement à chaque itération/récursion
  - annotation ajoutée au programme

- Spécification et mémoire

  > Un effet de bord désigne une intéraction d'une méthode avec la mémoire en dehors de ses arguments/valeur de retour (ex : la méthode lit une variable globale).

  Ceci peut poser des problèmes pour la vérification. Il faut donc expliciter ce que la méthode ne fait pas :

  - ajout du modificateur `pure` si la méthode n'a aucun effet de bord
  - sinon ajout de la clause `modifies` ou `assignable` pour dénoter les objets modifiés par la méthode

- Symbolic Execution Debugger

  > SED = debugger capable d'executer symboliquement un programme sans valeurs initiales

Étude de cas : le bug Timsort

En gros Timsort est un algorithme de tri hybride dérivé du tri fusion et du tri par insertion qui a été mis au point en 2002 par Tim Peters.

Cet algorithme de tri a été utilisé dans bon nombre de librairies que ça soit en Python, Java, Android, Rust, Swift... sauf qu'il présentait un bug. Une des méthodes de l'algorithme ne maintenait pas toujours l'invariant et on se retrouvait donc (dans certain cas) avec un `ArrayOutOfBoundsException`.

À l'aide du merveilleux logiciel KeY, Stijn de Gouw a pu :

- comprendre le bug
- produire des valeurs qui font crasher le programme
- proposer un fix

Et par la suite prouver que l'algorithme Timsort est bien correct en utilisant KeY.