Preuves de programmes : cours 1 (Introduction)

Frédéric Gava

ESIPE SI Ing2, Université de Paris-Est Créteil

Cours "Preuves de programmes" des Ing2 ESIPE SI UPEC

Plan

Plan

Déroulement du cours

En généralité

Notation

- Un devoir sur table ou TP noté
- Un exposé de papiers (par petits groupes)
- Examen final

Séances

- Voir ADE ; Prendre un PC portable
- 6 séances (normalement)
- On fera du JML et peut etre du SPIN
- Revoir les cours de Logiques, d'Algos et de Java

Introduction

Bug

Pourquoi ne veut on plus de bug ? Avantages et inconvénients

Généralités

- Rajouter des annotations/assertions/contrats pour la correction du code
- Correction partielle ou totale et fonctionelle
- Totale $\equiv$ partielle+terminaison
- Fonctionelle $\equiv$ qu'est-ce qui a été calculé ?
- Méthodes formelles $\equiv$ long long long

Outils

- Krakatoa, ESC/Java2, Key, Jack, VCG, *etc.* Aussi pour C, C#, *etc.*
- Génération d'obligations de preuves pour prouveurs (automatiques ou non) : Simplify, Z3, CVC3, Alt-Ergo, *etc.*

Introduction 2

Dans le cadre de la programmation orienté objets, nous avons des spécifications d'unités (contracts d'implementation) entre les applications et les bibliothèques. Les unités sont des interfaces, des classes et leurs méthodes. Nous commencerons par la spécification de méthodes avec :

1. Les valeurs initiales et les paramètres formelles
2. La valeur résultante
3. Les conditions initials (prestate) et final (poststate) aussi appelés pré- et post-conditions

Ceci va permettre de séparer les rôles (obligations) et responsibilités. Voir la spécification comme un contract ("Design by Contract" methodology, Meyer, 1992, langage Eiffel) entre l'appelé et l'appelant ("the called method").

Callee guarantees certain outcome

provided caller guarantees prerequisites

Exemple

```
public class ATM { (* NB: Carte bleue *)
//fields:
        private BankCard insertedCard = null;
        private int wrongPINCounter = 0;
        private boolean customerAuthenticated = false;
//methods:
        public void insertCard (BankCard card) { ... }
        public void enterPIN (int pin) { ... }
        public int accountBalance () { ... }
        public int withdraw (int amount) { ... }
        public void ejectCard() { ... }
}
```

Exemple informelle

Spécification fonctionelle et informelle de “enterPIN”

Enter the PIN that belongs to the currently inserted bank card into the ATM. If a wrong PIN is entered three times in a row, the card is confiscated. After having entered the correct PIN, the customer is regarded as authenticated.

Exemple par contracts

Un contract implique ce qui est garantie sous certaines conditions :

precondition : card is inserted, user not yet authenticated, pin is correct
postcondition : user is authenticated

precondition : card is inserted, user not yet authenticated, "wrongPINCounter$<$ 2" and pin is incorrect
postcondition : wrongPINCounter has been increased by 1, user is not authenticated

precondition : card is inserted, user not yet authenticated, "wrongPINCounter$>=$ 2" and pin is incorrect
postcondition : card is confiscated and user is not authenticated

Spécifications formelles

Les spécifications (fontionnelles) par des langages naturelles (anglais, etc.) sont très importantes et très utilisés mais nous allons nous concentrer sur les spécifications formelles :

Décrire les contrats avec une rigueure mathématique. Motivations :

- Meilleur degrè de précision
- Une formalisation peut mettre à jour des omissions (et/ou des étourderies)
- Enlève les ambiguités inhérentes aux langages naturels
- Analyse de programmes automatique :
  - Monitoring
  - Génération de cas de test
  - Vérification de programmes

Notion de Pre/Postcondition

Définition

Les pre/post-conditions pour une méthode “m” sont des propositions logiques satisfaisant l'implémentation de “m” si quand “m” est appelée alors pour chaque état de la machine (progamme) satisfaisant la pré-condition alors pour chaque état terminant de “m” la postcondition est aussi vrai.

Attention :

1. Rien n'est garanti quand la precondition n'est pas vrai
2. La terminaison de la méthode n'est pas garantie
3. Cette terminaison peut être normal ou abrupt (erreurs)

Nous verrons plus tard le problème de la terminaison.

Notion d'annotation logique (JML)

Notion de pre post conditions

{P}
code
{Q}

Si P est vrai, alors après l'execution du code, Q doit être vrai :

{x≤y} ou bien {pair(x)} ou bien {0≤x≤y}
x=x+1
{x≤y+1} ou bien {impair(x)} ou bien {0≤x≤y+1}

Donc, si on a {P}code{R} alors {P'}code{R'} si et seulement si P'⇒P et R⇒R'

Affectation

{P[x←e]} /* *substitution de x par e dans P* */
x=e
{P}

Cela est vrai si P[x←e]⇒x'=e⇒P[x←x']

Les cas d'annotations (2)

Exemple, affectation

{x+1>0}
x=x+1
{x>0}
Ici, $P[x \leftarrow e]$ est $x + 1 > 0$ et P est $x > 0$. Alors nous aurons :

x: **int**
H1: $x + 1 > 0$
x0: **int**
H2: $x0 = x + 1$
==========
$x0 > 0$

Une autre manière (parfois plus intuitive) de voir la chose est {P}x=e{Q} tel que Q[x←e]≡P. Alors, on aura {x≥0}x=x+1{x>0} et :

x: **int**
H1: $x \geq 0$
x0: **int**
H2: $x0 = x + 1$
==========
$x0 > 0$

Les cas d'annotations (3)

La séquence

Si on a {P1}code1{Q1} et {P2}code2{Q2}
alors on a {P1}code1;code2{Q2} si Q1⇒P2. Par exemple, {x≥0}x=x+1;x=x+1{x>1}. Donc :

x: **int**
H1: $x \geq 0$
x0: **int**
H2: $x0 = x + 1$
x1: **int**
H3: $x1 = x0 + 1$
==========
$x1 > 1$

Conditionnel (1)

Dans le cas d'un {P}**if** B then i1 **else** i2{S}, il est nécessaire les 2 branches. Nous aurons $\{P \wedge B\} i_1 \{S_1\}$ et $\{P \wedge \neg B\} i_2 \{S_2\}$ tel que $S \equiv S_1 \vee S_2$. Alors nous aurons à prouver :

1. $P \wedge B \implies S$
2. $P \wedge \neg B \implies S$

Les cas d'annotations (4)

Conditionnel (2), exemple

$\{0 \leq x < n\}$ **if** $(x < n-1)$ then $x = x+1$ **else** $x = 0$ $\{0 \leq x < n\}$. A prouver :

n: **int**
x: **int**
H1: $0 \leq x$ and $x < n$
H2: $x < n - 1$
x0: **int**
H3: $x0 = x + 1$
==========
$0 \leq x0 < n$

n: **int**
x: **int**
H1: $0 \leq x$ and $x < n$
H4: $x \geq n - 1$
x0: **int**
H5: $x0 = 0$
==========
$0 \leq x0 < n$

Conditionnel (3)

Notons, que nous avons fait un raccourci pour :
$\{0 \leq x < n\}$
if $(x < n-1)$ then $\{1 \leq x+1 < n\} x = x+1 \{1 \leq x < n\}$
else $\{0 \leq 0 < n\} x = 0 \{0 \leq x < n\}$
$\{0 \leq x < n\}$

La notion d'invariant (1)

Introduction

Les boucles (while, for, loop, *etc.*) nécessitent une annotation particulière : un **invariant**. En effet, on ne sait pas combien combien d'itérations aura la boucle et donc ce qu'elle modifiera :

while B **do**
{**invariant** P}
code
end

IMPORTANT : L'invariant P est une propriété qui est vrai au début de la boucle (avant d'y "rentrer") et à chaque itération. Ceci garantie que la propriété sera aussi vrai à la fin de la boucle.

Exemple

{i=0 $\wedge$ $n \geq 0$ $\wedge$ s=0}
while (i<n)
{**invariant** 2∗s=i∗(i+1)}
i=i+1; s=s+i
end
{2∗s=n∗(n+1)}

La notion d'invariant (2)

Nous aurons les buts suivants à prouver :

Loop invariant initialy holds

n: **int**
H1: $n \geq 0$
==========
$(2 \times 0 = 0 \times (0 + 1))$ and $0 \leq n$

The loop invariant preserved

n: **int** i: **int** s: **int**
H1: $n \geq 0$
H2: $2 \times s = i \times (i + 1)$ and $i \leq n$
H3: $i < n$
i0: **int**
H4: $i0 = i + 1$
s0: **int**
H5: $s0 = s + i0$
==========
$(2 \times s0 = i0 \times (i0 + 1))$ and $i0 \leq n$

The postcondition holds

n: **int** i: **int** s: **int**
H1: $n \geq 0$
H2: $2 \times s = i \times (i + 1)$ and $i \leq n$
H6: $i \geq n$
==========
$2 \times s = n \times (n + 1)$

Correction total

Terminaison : variant

Pour l'instant, nous pouvons prouver qu'un programme vérifie bien une propriété. Mais pas qu'il termine... Pour cela, nous rajoutons aux boucles un **variant**. C'est une mesure **strictement décroissante** et toujours **supérieur à 0**.

Exemple

{**invariant** 2∗s=i∗(i+1) variant n−i}. Ce qui nous donnera :

n: **int**
H1: $n \geq 0$
i: **int**
s: **int**
H2: $2 \times s = i \times (i + 1)$ and $i \leq n$
H3: $i < n$
i0: **int**
H4: $i0 = i + 1$
s0: **int**
H5: $s0 = s + i0$
==========
$(0 \leq n - i)$ and $(n - i0 < n - i)$

Label et ghost code

Label

Dans les annotations logiques, il est parfois utile d'exprimer la valeur à un endroit donner. On insére un label L:i=i+1 puis on utilise i@L c'.-à-d. la valeur de i avant l'affectation.

On écrit aussi i@ pour parler de la valeur i avant la méthode/procédure (en tant que paramètre).

Ghost code

Astuce. Des fois, il est impossible d'exprimer la propriété logique sans faire des calculs supplémentaires. On ajoute ces calculs dans le code et on parle alors de code fantôme : ce code peut lire les valeurs du programme en cours d'exécution mais PAS les modifier. On utilise ensuite les valeus du code fantôme dans les annotations.

Déroulement du cours

Java Modeling Language (JML)

Qu'est-ce ?

JML est le langage de spécification de Java (standard). Il contient une logique, des pre/postconditions, invariants, etc. (tout ne sera pas vu ici). Les annotations JML seront dans les codes sources (commentaires) afin de gagner en compatibilité mais elles sont ignorées par le compilateur.

Première méthode

```
public class Lesson1 {
    /*@ ensures \result >= x && \result >= y &&
      @ \forall integer z; z >= x && z >= y ==> z >= \result ;
      @*/
    public static int max (int x, int y) {if (x>y) return x; else return x /* erreur */;
}
```

"**ensures** "⇒post-condition de la méthode ; "**result** " la valeur de retour

Commentaires JML

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
@*/
public void enterPIN ( int pin) {
  if (...)
...
```

On peut aussi écrire

```
//@ public normal_behavior
//@ requires (!customerAuthenticated && (pin == insertedCard.correctPIN))
//@ ...
```

ATTENTION === /* @ ... */ Pas une annotation JML a cause du " ␣"

Comportement et “prestate”

Chaque mot clé terminant par behavior est un cas de specification.

Normal

normal_behavior $\Rightarrow$ la méthode est garantie pour ne pas lever d'exceptions si l'appelant garantie (à prouver !) toutes les preconditions pour ce cas de spécification.

Exemple

Pour notre exemple de code ATM, 2 préconditions (**requires**) :

1. !customerAuthenticated
2. pin == insertedCard.correctPIN

Résultat

La spécification ne comporte qu'une postcondition (**ensures**) qui est le boolean “customerAuthenticated”.

Differente specifications

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @
  @ also
  @
  @ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin != insertedCard.correctPIN;
  @ requires wrongPINCounter < 2;
  @ ensures wrongPINCounter == \old (wrongPINCounter)+1;
@*/
```

IMPORTANT : old(E) signifie “E” evaluée au moment du prestate, c’est-à-dire ici au moment de l’appelant de la méthode “enterPIN”.

“E” peut être n’importe quelle expression JML (même complexe !)

Differente specifications

```
/*@ <spec-case1> also <spec-case2> also
  @
  @ public normal_behavior
  @ requires insertedCard != null ;
  @ requires !customerAuthenticated;
  @ requires pin != insertedCard.correctPIN;
  @ requires wrongPINCounter >= 2;
  @ ensures insertedCard == null;
  @ ensures \old (insertedCard).invalid;
@*/
public void enterPIN ( int pin) { ...
```

Deux post-conditions signifie que les 2 expressions sont toutes les 2 valides (vrais) après l'exécution de la méthode (un "and==&&").

Est-ce que "**ensures** **old** (insertedCard.invalid)" entraine nécessairement "**ensures** **old** (insertedCard).invalid" ? ? ?

On complète !

@ **public normal_behavior** (∗ <spec−case1> ∗)
@ **requires** !customerAuthenticated;
@ **requires** pin == insertedCard.correctPIN;
@ **ensures** customerAuthenticated;

On ne sait rien sur “insertedCard” ni sur “wrongPINCounter” ! ! !

On complète :

@ **ensures** insertedCard == **old** (insertedCard);
@ **ensures** wrongPINCounter == **old** (wrongPINCounter);

De même pour “spec-case2” :

@ **ensures** insertedCard == **old** (insertedCard);
@ **ensures** customerAuthenticated == **old** (customerAuthenticated);

et “spec-case3” :

@ **ensures** customerAuthenticated == **old** (customerAuthenticated);
@ **ensures** wrongPINCounter == **old** (wrongPINCounter);

Et pour ce qui n'est pas modifié ?

On n'a pas envie d'écrire :

@ **ensures** champs==**old** (champs);

pour tout les "champs" qui ne sont pas modifiés par la méthode.
On écrit :

@ **assignable** loc1,...,locn;

Pour signifier que seuls ces champs seront modifiés (sous-entendu, les autres ne le sont pas). Aussi :

@ **assignable** **nothing** ;
OU BIEN (∗ a éviter ∗)
@ **assignable** **everything** ;
MAIS AUSSI !
@ **assignable** o.∗ , a[∗];
=== pour tout les champs de l'objet "o" et pour toutes les indices du tableau "

Exemple de "spec-case3" :

@ **public normal_behavior**
@ **requires** ...;
@ **ensures** ...;
@ **assignable** insertedCard, insertedCard.invalid;

JML Modificateurs d'accès

Les "Modifiers" les plus importants

spec_public , **pure** , **nullable** , **non_null** , **helper** , etc.

Dans notre précèdents exemples, nous utilisions massivement les champs. Mais les spécifications **public** ne peuvent accéder qu'à des données **public** (pour des raisons de sécurité). Nous ne voulons pas tout avoir en **public** .

Contrôler la visibilité avec spec_public

On peut accéder aux champs **private** /**protected** et si nécessaire, rendre **public** que dans les spécifications.

private /*@ **spec_public** @*/ BankCard insertedCard = **null** ;
private /*@ **spec_public** @*/ **int** wrongPINCounter = 0;
private /*@ **spec_public** @*/ **boolean** customerAuthenticated = **false** ;

Pureté (1)

On peut utiliser des appels de méthodes dans les annotations JML comme :

o1.equals(o2) OU li.contains(elem) OU li1.**max** () $<$ li2.**min** ()

Mais ces appels ne doivent pas changer eux-meme l'état en cours.

Les méthodes pures

Une méthode est pure si et seulement si elle termine toujours et n'a pas d'effet sur les objets. Une méthode est strictement pure si de plus elle ne créé pas d'objets.

Les annotations ne peuvent contenir que des fonctions pures.

public /∗@ **pure** @∗/ **int max** () { ... }

Pureté (2)

- **pure** oblige le programmeur a ne pas faire d'effet de bords
- Il est possible de vérifier cela formellement
- **pure** implique **assignable** **nothing** (mais peut créer de nouveaux objets temporairement)
- **assignable** **strictly_nothing** pour exprimer qu'aucun objet ne sera créé
- Attention ! Ce genre de clauses sont locales aux cas de spécification
- tandis que **pure** est globale à la méthode

JML Expressions $\neq$ Java Expressions

- Chaque expression boolean Java sans effet de bord est une expression JML.
- Si “a” et “b” sont des expressions boolean JML et si x est une variable de type “t” alors ce qui suient sont aussi des expressions boolean :
  - !a (non a)
  - (a && b), (a || b), (a==>b) et (a <==>b) (et, ou, implique, equivalent)
  - Les prédicats de rang (quantificateurs) comme :
  - (**forall** t x; a) (pour tout x de type t, a est vrai)
  - (**exists** t x; a) (il existe x de type t tel que a est vrai)
  - (**forall** t x; a; b) (pour tout x de type t tel que a est vrai alors b est vrai). Sucre syntaxique de (**forall** t x; a ==> b)
  - (**exists** t x; a) (il existe x de type t tel que a est vrai alors b est vrai). Sucre syntaxique de (**exists** t x; a && b)
  - ...

Exemples

- "Un tableau arr est trié de l'indice 0 à 9".
  (**forall int** i,j; 0<=i && i<j && j<10; arr[i] <= arr[j])
- "Un tableau ne contenant que des entiers inférieurs a 2
  (**forall int** i; 0 <= i && i < arr.length; arr[i] <= 2)
- "Le contenu de la variable m est le max du tableau"
  (**forall int** i; 0 <= i && i < arr.length; m >= arr[i])
  && (arr.length > 0 ==>
  (**exists int** i; 0 <= i && i < arr.length; m == arr[i]))
- "Toutes les instances existantes de la class BankCard ont des CardNumbers différents"
  (**forall** BankCard p1, p2; p1!=p2 ;
  p1.cardNumber != p2.cardNumber)

D'autres quantificateurs

JML offre d'autre quantificateurs comme **max** , **min** , **product** , **sum** qui retournent respectivement les maximums, minimums, produits et sommes d'expressions au rangs bien définies. Exemples :

- (**sum int** i; 0<=i && i<5; i) $= 0 + 1 + 2 + 3 + 4 + 5$
- (**product int** i;0<i && i<5; (2*i)+1) $= 3 * 5 * 7 * 9$
- (**max int** i;O <=i && i<5; i) $= 4$

Exemple : spécifier un ensemble d'entiers

```
public class LimitedIntegerSet {
  public final int limit;
  private int arr[];
  private int size = 0;

  public LimitedIntegerSet( int limit) {
    this .limit = limit;
    this .arr = new int [limit];
  }

  public boolean add( int elem) {/*...*/}

  public void remove( int elem) {/*...*/}

  public boolean contains( int elem) {/*...*/}

  // other methods
}
```

Exemple : spécifier un ensemble d'entiers

```
public class LimitedIntegerSet {
  public final int limit;
  private /*@ spec_public @*/ int arr[];
  private /*@ spec_public @*/ int size = 0;

  public LimitedIntegerSet( int limit) {
    this .limit = limit;
    this .arr = new int [limit];
  }

  public boolean add( int elem) {/*...*/}

  public void remove( int elem) {/*...*/}

  public /*@ pure @*/ boolean contains( int elem) {/*...*/}
    ===> Il va falloir ici spécifier un résultat
  // other methods
}
```

Valeur de retour dans une post-condition

Retour

Dans une post-conditon, \result permet de référer à la valeur de retour de la méthode. Exemple :

```
/*@ public normal_behavior
  @ ensures \result == ( \exists int i; <= i && i < size; arr[i] == elem);
@*/
public /*@ pure @*/ boolean contains( int elem) {/*...*/}

/*@ public normal_behavior (* <spec-case1>, ajout d'un nouvel élément *)
  @ requires size < limit && !contains(elem);
  @ ensures \result == true ;
  @ ensures contains(elem);
  @ ensures ( \forall int e; e != elem; contains(e) <==> \old (contains(e)));
  @ ensures size == \old (size) + 1;
  @ also
  @ <spec-case2>
@*/
public boolean add( int elem) {/*...*/}
```

Valeur de retour dans une post-condition (suite)

```
/*@ public normal_behavior
  @ <spec-case1>
  @ also
  @ public normal_behavior (* ajout non effectif *)
  @ requires (size == limit) || contains(elem);
  @ ensures \result == false ;
  @ ensures ( \forall int e; contains(e) <==> \old (contains(e)));
  @ ensures size == \old (size);
@*/
public boolean add( int elem) {/*...*/}

/*@ public normal_behavior
  @ ensures !contains(elem);
  @ ensures ( \forall int e; e != elem; contains(e) <==> \old (contains(e)));
  @ ensures \old (contains(elem)) ==> size == \old (size) - 1;
  @ ensures !\old (contains(elem))==> size == \old (size);
@*/
public void remove( int elem) {/*...*/}
```

Spécifier des contraintes de données (1)

JML ne sert pas qu'à spécifier des méthodes.

Contraintes de class, pourquoi ?

- Consistence de la représentation redondante de données
- Restriction pour l'efficacité (e.g. trié)
- Toutes les méthodes se doivent de respecter ces contraintes

Par exemple, pour insérer un ensemble représenté par un tableau trié, on peut utiliser un recherche logarithmique, mais que possible si véritablement trié (prestate). De même, il faut garantir que cela reste trié (potstate) pour un projet ajout/recherche.

Mettre cette propriété de "trié" partout (pre/postconditions) peut s'avéré laborieux.

Spécifier des contraintes de données (2)

JML invariant de classe

Pour centraliser la spécification des contraintes de class (structure de données). Exemple :

```
public class LimitedSortedIntegerSet {
  public final int limit;
  /*@ private invariant ( \forall  int i; 0 < i && i < size; arr[i−1] <= arr[i]);
    @*/
  private /*@ spec_public @*/ int arr[];
  private /*@ spec_public @*/ int size = 0;
  // constructeurs et methodes,
  // n'ont plus besoin de spécifier le tri des entiers dans les pre/postconditions
```

On peut mettre les invariants de classes où on veut mais au moins devant les champs qu'ils manipulent.

Invariant d'instance vs static

Instance (par défaut pour les classes !)

Peut référer aux instances des champs (**this**) : **instance invariant**

Static (par défaut pour les interfaces !)

Ne peux pas se référer aux instances de l'objet : **static invariant**

Pour les 2, on peut se référer aux autres champs statiques ainsi qu'aux instances d'autres objets.

```
public class BankCard {
 /*@ public static invariant
   @ (\forall BankCard p1, p2; p1 != p2 ==> p1.cardNumber != p2.cardNumber)
   @*/
  private /*@ spec_public @*/ int cardNumber;
  // reste de la classe
}
```

Les invariants doivent être vrai pour les constructeurs (instance), pour l'initialisation statique. Assumé vrai pour les

Contre-ordre

Les méthodes "helper" en

T /*@ **helper** @*/m(T p1, ..., T pn) pour ne pas assumer ni assurer aucun invariant **par défaut**.

Mais pourquoi ? ? ? Pragmatisme...

- Généralement privées (internes)
- Pour les constructeurs (quand des invariants ne sont pas encore établis).

Référer des invariants

Rôle

But : référer des invariants d'autres objets dans les expressions JML : **invariant_for** (o) est vrai dans un état quand **tout** les invariants de o sont vrai, sinon c'est faut.

Pragmatisme :

- On utilise **invariant_for** (**this**) quand les invariants locaux sont attendus mais pas donnés explicitement (pour les méthodes **helper**).
- Mettre **invariant_for** (o) (o≠**this**) pour les **requires** /**ensures** locaux ou dans les invariants pour admettre/assurer ou maintenir les invariants de o localement.

Exemple 1

Si toutes les méthodes (non-helper) d'ATM doivent maintenir les invariants des objets stockés avec insertedCard :

```
public class ATM {
  ...
  /*@ private invariant
    @ insertedCard != null ==> \invariant_for (insertedCard);
    @*/
  private BankCard insertedCard;
  ...
  /*@ public normal_behavior
    @ requires \invariant_for  (insertedCard);
    @ requires <other preconditions> ;
    @ ensures <postcondition> ;
  @*/
  public int withdraw ( int amount) { ... }
  ...
```

Exemple 2

```
public class Database {
  ...
  /*@ public normal_behavior
    @ requires ...;
    @ ensures ...;
    @*/
  public void add (Set newItems) {
    ... <rough adding at first> ...;
    cleanUp();
  }
  ...
  /*@ private normal_behavior
    @ ensures \invariant_for (this);
    @*/
  private /*@ helper @*/ void cleanUp() { ... }
```

Référer des invariants (2)

- Pour des méthode “non-helper”, **invariant_for** (**this**) est ajouté implicitement aux pre-postconditions
- **invariant_for** (expr) retourne vrai que si et seulement si expr satisfait les invariants de son propre type statique (sa classe actuelle) :
  - Assumons **class** B **extends** A
  - Après l'exécution des “initialiseurs”, A o=**new** B(); alors **invariant_for** (o) est vrai quand o satisfait les invariants de A mais **invariant_for** ((B)o) est vrai quand o satisfait les invariants de B !
- Si o et **this** ont des types différents alors **invariant_for** (o) ne couvre que les invariants publics du type de o. Par exemple, **invariant_for** (insertedCard) ne refère qu'aux invariants publics de BankCard.

Exception et comportement anormal (1)

Rappel sur la spécification de enterPIN :

private /*@ **spec_public** @*/ BankCard insertedCard = **null**;
private /*@ **spec_public** @*/ **int** wrongPINCounter = 0;
private /*@ **spec_public** @*/ **boolean** customerAuthenticated=**false**;

/*@ <*spec−case1*> **also** <*spec−case2*> **also** <*spec−case3*> @*/
public void enterPIN (**int** pin) { ... }

Les 3 “spec-cases” étaient en **normal_behavior** . On interdit donc aux méthodes de lever des exceptions si la précondition est satisfaite. On souhaite bien sûr pouvoir indiquer les cas problématiques (exceptions).

signals permet de spécifier un “poststate” dépendant de l'exceptions levée et **signals_only** limite les types à cette exception.

Exception et comportement anormal (1)

```
/*@ <spec-case1> also <spec-case2> also <spec-case3> also
@ public exceptional_behavior
@ requires insertedCard==null;
@ signals_only ATMException;
@ signals (ATMException) !customerAuthenticated;
@*/
public void enterPIN ( int pin) { ...
```

Dans le cas où insertedCard==**null** alors :

- enterPIN doit lever une exception ("exceptional_behavior")
- Cela ne peut que être qu'une ATMException ("signals_only")
- Comme signaler dans le poststate, la méthode doit assurer que !customerAuthenticated (non pas authentifié)

En général

Signals Only

Dans les cas d'exceptions, on peut un : **signals_only** E1,...,En où les Ei sont des types d'exception afin d'indiquer que s'il y a une exception elle est forcement d'un de ces types. Par défaut Ei sont TOUTES les exceptions déclarées dans **throws**+RuntimeException+Error.

Signals Only

On peut écrire plusieurs clauses : **signals** (E) b; où b est une expression boolean afin d'indiquer que si l'exception levé est de type E alors b est vrai (postcondition).

Exemple

```
class NoCreditException extends Exception {
  /*@ assigns \nothing ; */
  public NoCreditException(){} }

public class Purse2 {
 /*@ public normal_behavior
   @ requires s >= 0;
   @ assigns balance;
   @ ensures s <= \old (balance) && balance == \old (balance)−s;
   @ behavior amount_too_large:
   @ assigns \nothing ;
   @ signals (NoCreditException) s > \old (balance) ;
   @*/
  public void withdraw2(int s) throws NoCreditException {
    if (balance >= s) {balance = balance − s;}
     else {throw new NoCreditException();}
```

“**signals** ” permet d’exprimer les cas d’exception. Avec “behavior”, on donne un nom à ce cas d’exception. Un “behavior” est le nom d’un cas. On pourrait avoir “behavior comportement_normal: ...”

Vers l'infini et au delà !

Par défaut, les **normal_behavior** et exceptional_behavior sont des cas qui forcent la terminaison. Pour un serveur par exemple, on peut indiquer qu'un cas diverge (ne termine pas) avec la clause

diverges true

Attention donc, suivant les cas spécification dans la precondition, la méthode peut ou non terminer...Et si elle ne termine pas, il n'y a pas de valeur de retour !

T'es null, T'es bat, t'es in

On peut spécifier que des valeurs sont **nullable** (potentiellement **null**) ou **non_null** (ne peuvent pas être **null**). Exemple :

public void insertCard(/*@ **non_null** @*/ BankCard card) {...}
===> Pour chaque cas de spécification de la méthode, on ajoute la precondition implicite
requires card!=**null**;

public /*@ **non_null** @*/ String toString()
===> Pour cas cas de spécification de la méthode, on ajoute la postcondition implicite
ensures **result** !=**null**;

private /*@ **spec_public non_null** @*/ String name;
===> On ajoute à la classe l'**invariant** implicite
public invariant name!=null;

non_null est **recemment** (grrr) par défaut dans JML ! Donc ici inutile mais cela peut aider à la compréhension de la spécification.

T'es null, T'es bat, t'es in

Attention :

public void insertCard(/*@ **nullable** @*/ BankCard card) {...}
===> Rien d'implicite d'ajouter

public /*@ **nullable** @*/ String toString()
===> Rien d'implicite d'ajouter

private /*@ **spec_public nullable** @*/ String name;
===> Rien d'implicite d'ajouter

Cela évite ces ajouts implicites. Exemple :

public class LinkedList { | **public class** LinkedList {
private Object elem; | **private** Object elem;
private LinkedList next; | **private** /*@ **nullable** @*/ LinkedList next;
... |
|
Elements non **null** mais liste | Liste peut finir quelque part
infini ou cyclique |

Pourquoi grrrr ?

Car maintenant :

- /*@ *non null* @*/Object[] a; n'est pas la meme chose que /*@ **nullable** @*/Object[] a; //@ **invariant** *a != null;*
- ???? Car dans la premire on ajoute maintenant implicitement (**forall int** i; i >= 0 && i < a.length; a[i] != **null**) i.e. les élements du tableau sont aussi non null.

JML et héritage

Tout les contracts JML (cas de spécification, invariant de classes) héritent (de super-classe a sous-classe). Donc une classe fille hérite de tout les contracts de sa classe mère. Et on peut bien sûr rajouter des cas de spécification supplémentaire :

```
/*@ also
  @
  @ <subclass-specific-spec-cases>
  @*/
  public void method () { ... }
```

Il existe encore d'autres mot clés en JML comme callable, captures, working_space, duration, accessible, when. Plus de détails dans les manuels techniques.

Rapidement

- \assignale A; Seul les allocation dans le heap (static,champs d'instance, tableaux) qui n'existaient pas dans la précondition (prestate) ou qui ont été listé en A peuvent avoir été changé
- **invariant_for** (**this**) doit être maintenu par les les methodes non-helper

Dans le cas du **normal_behavior** on a par défaut **signals** (Throwable e) **false**; et on interdit les signal et **signals_only** . Dans un exceptional_behavior, on a par défaut **ensures false** et les outils généralement interdisent les **ensures** .

Des lectures

1. KeYbook. W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. Schmitt, M. Ulbrich, editors. Deductive Software Verification - The KeY Book Vol 10001 of LNCS, Springer, 2016
2. JML Tutorial M. Huisman, W. Ahrendt, D. Grahl, M. Hentschel. Formal Specification with the Java Modeling Language Chapter 7
3. www.eecs.ucf.edu/ leavens/JML//index.shtml

A la semaine prochaine