Assertions avec JML

Marc Champesme & Guillaume Vauvert
Département d'informatique – Institut Galilée
Université Paris 13

22 mars 2003

On peut inclure trois types d'assertions : les invariants, associés à la classe, les preconditions et les postconditions, toutes deux associées à des méthodes. Avec JML, les assertions s'écrivent sous forme de commentaires situés dans le code de la classe ou avant l'entête des méthodes, à des emplacements précis et selon une syntaxe précise. Le but de ce document est de décrire cette syntaxe.

1 Syntaxe commune à tous les types d'assertion

Quelque soit le type d'assertion (invariant, précondition ou postcondition), certaines règles syntaxiques sont identiques :

- Le commentaire contenant l'assertion est délimité par les caractères `/*@` qui indiquent le début du commentaire et par `@*/` pour indiquer la fin du commentaire. Lorsque les assertions s'étendent sur plus d'une ligne, il est recommandé, afin d'assurer une meilleure lisibilité, de débuter chaque ligne par le caractère `@`.
- Le commentaire contenant l'assertion doit être placé en dehors du commentaire au format javadoc, l'emplacement recommandé est immédiatement après le commentaire javadoc c'est-à-dire, entre le commentaire javadoc et l'entête de la méthode.
- Lorsque l'assertion occupe une seule ligne, il est possible d'utiliser le délimiteur de commentaire `\\@`. Dans ce cas, le commentaire se termine en fin de ligne et il n'y a pas de délimiteur de fin de commentaire.
- Après la séquence de caractères indiquant le début du commentaire, un mot-clé indique le type d'assertion : `invariant` pour un invariant, `requires` pour une précondition et `ensures` pour une postcondition.
- Juste après le mot-clé indiquant le type d'assertion, il est possible de placer un label indiquant le rôle de cette assertion. Ce label apparaîtra dans le message affiché si JML détecte une violation d'assertion lors de l'exécution du programme, l'intérêt de ce label est donc d'aider le programmeur à détecter la source de l'erreur diagnostiquée par cette violation d'assertion. D'autre part, ce label apparaîtra dans la documentation de la classe générée par javadoc. La syntaxe à utiliser pour donner un label à une assertion est la suivante :
  - `\lblneg( label assertion );`
  - la chaîne de caractères constituant le label doit respecter les mêmes règles que celles définies pour les identificateurs JAVA : en particulier, un label ne doit pas contenir le caractère espace.
- Chacune des expressions booléennes constituant une assertion doit être terminée par un point-virgule. Du point de vue de la signification de l'assertion, ce point-virgule est équivalent à un et logique (opérateur JAVA `&&`).
- Lorsqu'il n'est pas possible d'exprimer l'assertion sous forme d'une expression JAVA, l'assertion pourra être spécifiée uniquement par un commentaire délimité par `(*` et `*)` (il s'agit d'un commentaire multiligne).

2 Invariant de classe

Il doit être placé dans un commentaire situé juste après la déclaration de la classe (i.e. la ligne `public class IdentificateurDeClasse {`.

Exemple 2.1 Exemple d'invariant

```
/**
 * Cette classe permet d'effectuer des calculs sur les nombres complexes ...
 *
 * @author Moi
 * @created 18/12/2001
 * @version 0.1
 */
public class Complexe {
    /*@ invariant getNorme() == Math.sqrt(getPartieReelle()*getPartieReelle()
      @                             + getPartieImaginaire()* getPartieImaginaire());
      @*/
    ...
    //@ pure
    public double getPartieReelle() {
        ...
    }
    ...
    //@ pure
    public double getPartieImaginaire() {
        ...
    }
    ...
    //@ pure
    public double getNorme() {
        ...
    }
    ...
}
```

3 Préconditions

Les préconditions se placent juste avant l'entête des méthodes.

Exemple 3.1 Exemple de précondition

```
    /**
      * Permet de calculer ...
      * @param x paramètre utilisé pour calculer ...
      * @return le résultat du calcul ...
      */
    /*@ pure
      @ requires \lblneg( x_existe  x != null);
      @ requires x.getNorme() != 0;
      @*/
    public double getResultatDuCalcul(Complexe x) {
        // Le code de la méthode
    }
```

4 Postconditions

Les postconditions se placent juste avant l'entête des méthodes, au même emplacement que les préconditions.

Exemple 4.1 Méthode qui donne une valeur à une propriété :

```
    /**
     * Permet de donner une nouvelle valeur à  ...
```

```
 * @param x La nouvelle valeur de ...
 */
/*@
  @ requires    \lblneg( x_existe  x != null);
  @ ensures     getValue() == x;
  @*/
public void setValue(Object x) {
    // Le code de la méthode
}
```

Exemple 4.2 Méthode qui retourne une valeur :

```
/**
 * Renvoie ...
 * @param x Nombre complexe ...
 * @return La valeur de ...
 */
/*@ pure
  @ requires    \lblneg( x_existe  x != null);
  @ requires    x.getNorme() != 0;
  @ ensures     \result == (x.getNorme() * getDeviation());
  @*/
public double getNormalDeviation(Complex x) {
    // Le code de la méthode
}
```

4.1 Faire référence au résultat de la méthode : la variable `result`

Pour les méthodes renvoyant un résultat, il est souvent nécessaire de faire référence à ce résultat dans les postconditions de la méthode. Dans ce but, JML met à notre disposition la variable `\result`. Cette variable possède les caractèristiques suivantes :

- La valeur de cette variable est la valeur de l'expression spécifiée dans l'instruction `return` de la méthode.
- Le type de cette variable est le type déclaré pour la valeur de retour de la méthode.
- La variable `\result` est utilisable *uniquement* pour l'écriture des postconditions, elle n'est pas définie en dehors ce contexte. Elle ne peut donc être utilisée, ni dans le code JAVA de la méthode, ni dans les autres types d'assertion.

Exemple 4.3 Utilisation de la variable `\result`

```
/**
 * Renvoie la norme de l'instance.
 */
/*@ pure
  @ ensures \result == Math.sqrt(getPartieReelle()*getPartieReelle()
  @                         + getPartieImaginaire()* getPartieImaginaire();
  @*/
public void getNorme() {
    ...
}
```

4.2 Faire référence à l'état antérieur de l'instance : opérateur `old(...)`

Dans l'écriture des postconditions, il est souvent nécessaire de faire référence à l'état antérieur de l'instance. Dans ce but, JML met à notre disposition l'opérateur `\old(..)`. Cet opérateur possède les caractèristiques suivantes :

- La valeur d'une expression `\old(..)` est calculée par JML immédiatement avant l'exécution du code de la méthode.

– Une expression `\old(..)` est utilisable *uniquement* pour l'écriture des postconditions, elle n'est pas définie en dehors ce contexte. Elle ne peut donc être utilisée, ni dans le code JAVA de la méthode, ni dans les autres types d'assertion.

Exemple 4.4 Utilisation d'une expression `\old(..)` :

```
public class Pile {
    ...
    /**
     * Place l'objet passé en paramètre sur la pile.
     * @param o L'Objet à mettre sur la pile.
     */
    /*@
      @ ensures getSize() == \old(getSize()) + 1;
      @*/
    public void empiler(Object o) {
        ...
    }
}
```

5 Quel code a-t-on le droit de mettre dans une assertion ?

5.1 Prise en compte de la visibilité des caractéristiques

Une assertion est une expression booléenne destinée à être évaluée. Il faut que toutes ses variables :
– soient déclarées et visibles au moment de leur vérification, ce qui exclu les variables locales à une méthode pour les préconditions et les invariants ;
– soient visibles par l'utilisateur de la classe (sinon, l'assertion n'a aucun sens pour lui), ce qui exclu les propriétés privées (et généralement, toutes les propriétés sont déclarées privées, ce qui oblige à utiliser des méthodes du type `getX()`, `getValue()`, ...), et les variables locales à une méthode (déclarées dans le corps).

Exemple 5.1 Ce qu'il ne faut pas faire :

```
public class Complexe {
    private double partieReelle;
    private double partieImaginaire;
    /**
     * Calcule et renvoie le résultat de la division de l'instance courante par un
     * autre nombre complexe.
     * @param z Le nombre complexe qui divise le complexe courant.
     * @result Le résultat de la division de l'instance courante par un
     * autre nombre complexe.
     */
    /*@ pure
      @ requires    partieReelle != 0;
      @ requires    partieImaginaire != 0;
      @     ...
      @ requires    normeZ != 0;
      @*/
    public Complexe diviserPar(Complexe z) throws ComplexeDivisionParZero {
        double normeZ = z.getNorme();
        if (normeZ == 0) {
            throw new ComplexeDivisionParZero();
        } else {
            return (this.multiplierPar(z.conjugue())).produitScalairePar(1/normeZ);
        }
    }
```

```
    ...
}
```

A corriger par :

```
public class Complexe {
    private double partieReelle;
    private double partieImaginaire;
    ...
    //@ pure
    public double getPartieReelle() {
        return partieReelle;
    }

    //@ pure
    public double getPartieImaginaire() {
        return partieImaginaire;
    }

    //@ pure
    public double getNorme() {
        ...
    }

    /**
     * Calcule et renvoie le résultat de la division de l'instance courante par un
     * autre nombre complexe.
     * @param z Le nombre complexe qui divise le complexe courant.
     * @result Le résultat de la division de l'instance courante par un
     * autre nombre complexe.
     */
    /*@ pure
      @ requires    getPartieReelle() != 0;
      @ requires    getPartieImaginaire() != 0;
      @     ...
      @ requires    z.getNorme() != 0;
      @*/
    public Complexe diviserPar(Complexe z) throws ComplexeDivisionParZero {
        double normeZ = z.getNorme();
        // Le test sur la valeur de la norme de z, n'a pas sa place ici puisque
        // la précondition joue justement ce rôle.
        return (this.multiplierPar(z.conjugue())).produitScalairePar(1/normeZ);
    }
    ...
}
```

5.2 Méthodes sans effets de bord : utilisation du mot-clé `pure`

Les assertions utilisant souvent des appels de méthodes, il convient de s'assurer que les méthodes utilisées sont sans effet de bord. A cet effet, JML introduit le mot-clé `pure` pour qualifier de telles méthodes. Ce mot clé doit être utilisé de la manière suivante :

- Avec JML, seules les méthodes spécifiée *pure* peuvent-être utilisées dans les assertions.
- Toute méthode sans effet de bord doit être qualifiée de *pure*. En effet, même si cette méthode n'est pas utilisée dans l'écriture des assertions de la classe courante, il faut laisser le plus de souplesse possible au programmeur d'autres classes qui utiliseraient la classe courante et pourraient donc avoir besoin d'utiliser les méthodes de la classe courante pour écrire les assertions de ces nouvelles classes.

6 Autres opérateurs de JML

La grammaire de JML pour les assertions est un peu plus riche que celle de Java. En effet, en plus des expressions booléennes standards de Java, JML permet de construire des expressions booléennes utilisant des quantificateurs universels ou existenciels.

6.1 Quantificateurs universel et existentiel

`\forall` (quelque soit) est un quantificateur universel et `\exists` (il existe) est un quantificateur existentiel.

Exemple 6.1 Expression quantifiée universellement.

```
(\forall int i, j ; 0 <= i && i < j && j < 10 ; tab[i] < tab[j])
```

spécifie que les éléments du tableau `tab` dont l'indice est compris entre 0 et 9 sont triés par ordre croissant.

Les variables quantifiées (i.e. `i` et `j` dans l'exemple 6.1) prennent toute les valeurs possibles satisfaisant l'expression booléenne située en deuxième partie de l'expression quantifiée entre les points-virgule (`;`) (i.e. `0 <= i && i < j && j < 10` dans l'exemple 6.1). Si cette expression booléenne est absente, sa valeur par défaut est l'expression `true` (i.e. toujours vraie) et l'expression porte sur toutes les valeurs possibles des variables quantifiées. Le type d'une expression quantifiée est `boolean`.

```
(\forall <type> <var1>, <var2>, ... ; <expr_valeurs_possibles> ; <Expr_var>);
(\forall int <var> ; <borne_inf> <= <var> && <var> <= <borne_sup> ; <Expr_var>);
(\exists <type> <var1>, <var2>, ... ; <expr_valeurs_possibles> ; <Expr_var>);
(\exists int <var> ; <borne_inf> <= <var> && <var> <= <borne_sup> ; <Expr_var>);
```

Exemple 6.2 Exemple avec une assertion `exists` et une assertion `forall` :

```
public class TestJml {
    public int[] tab;

    public static void main(String[] args) {
        System.out.println("Création d'une instance:");
        TestJml obj = new TestJml();
        System.out.println("Instance créée avec succès: Bye...");
    }
    /*@
      @ ensures (* Déclenche une violation de postcondition si
      @            un des éléments du tableau est égal a 2 : *) &&
      @     (\lblneg
      @         aucun_2
      @         (\forall int i ; 0 <= i && i < tab.length ; tab[i] != 2) );
      @ ensures (* Au moins un des éléments du tableau doit être égal a 3 : *) &&
      @     (\lblneg
      @         au_moins_un_3
      @         (\exists int i ; 0 <= i && i < tab.length ;  tab[i] == 3) );
      @*/
    public TestJml() {
        tab = new int[10];
        tab[2] = 2; // Violation de l'exception [aucun_2]
    }
}
```

6.2 Quantificateurs généralisés

Les quantificateurs `\max`, `\min`, `\product` et `\sum` sont des quantificateurs généralisés qui renvoient, respectivement, le maximum, le minimum, le produit ou la somme des valeurs de l'expression donnée, quand les variables satisfont l'expression déterminant les valeurs possibles. L'expression déterminant les

valeurs possibles doit être de type `boolean`. L'expression dont les valeurs sont prises en compte pour le calcul doit être d'un type de base numérique comme `int` ou `double` ; le type de l'expression quantifiée est du même type que cette dernière expression. L'expression dont les valeurs sont prises en compte pour le calcul est la dernière expression, c'est-à-dire celle suivant immédiatement l'expression déterminant les valeurs possibles. A titre d'exemple, les expressions suivantes sont toutes vraies :

```
(\sum int i; 0 <= i && i < 5; i) == 0 + 1 + 2 + 3 + 4
(\product int i; 0 < i && i < 5; i) == 1 * 2 * 3 * 4
(\max int i; 0 <= i && i < 5; i) == 4
(\min int i; 0 <= i && i < 5; i-1) == -1
```

Exemple 6.3 Exemple d'utilisation des quantificateurs généralisés :

```
public class TestJml {
    public int[] tab;
    public int[] autreTab;

    public static void main(String[] args) {
        System.out.println("Création d'une instance:");
        TestJml obj = new TestJml();
        obj.testQuantificateurGeneralise();
        System.out.println("Instance créée avec succès: Bye...");
    }

    public TestJml() {
        tab = new int[10];
        tab[2] = 3;
        tab[3] = 1;
        tab[4] = 3;
    }

    /*@
      @ requires (\lblneg
      @             min_0
      @             (\min int i ; 0 <= i && i < tab.length ;  tab[i]) == 0 );
      @ requires (\lblneg
      @             max_3
      @             (\max int i ; 0 <= i && i < tab.length ;  tab[i]) == 3 );
      @ ensures (\lblneg
      @             somme_sup_tab_length
      @             (\sum int i ; 0 <= i && i < tab.length ;  tab[i]) > tab.length );
      @ ensures (\lblneg
      @             produit_sup_4
      @             (\product int i ; 0 <= i && i < tab.length ;  tab[i]) > 4 );
      @ ensures (\lblneg
      @             min_1
      @             (\min int i ; 0 <= i && i < tab.length ;  tab[i]) == 1 );
      @*/
    public void testQuantificateurGeneralise() {
        for (int i = 0; i < tab.length; i++) {
            tab[i] = 1;
        }
        tab[5] = 3;
        tab[6] = 2;
    }
}
```

6.3 Le mot-clé `fresh`

L'opérateur `\fresh` spécifie que des objets ont été alloués depuis le début de l'exécution de la méthode. Cet opérateur ne doit donc être utilisé que dans des postconditions. Par exemple, l'assertion `\fresh(x, y)` spécifie que les variables `x` et `y` ne sont pas `null` et qu'elle contiennent des références à des objets qui n'étaient pas encore créés au début de l'exécution de la méthode. Les arguments de `\fresh` peuvent être de n'importe quel type objet et le type d'une expression `\fresh(...)` est boolean.

Exemple 6.4 Exemple d'utilisation de `\fresh(...)` :

```
public class TestJml {
    public int[] tab;
    public int[] autreTab = new int[10];

    public static void main(String[] args) {
        System.out.println("Création d'une instance:");
        TestJml obj = new TestJml();
        System.out.println("Instance créée avec succès: Bye...");
    }
    /*@
      @ ensures \fresh(tab);
      @ ensures \lblneg(
      @             nouveau_autreTab
      @             \fresh(autreTab) );
      @*/
    public TestJml() {
        tab = new int[10];
        tab[2] = 2;
        // Violation de l'assertion \fresh(autreTab)
    }
}
```

Note : L'utilisation de l'expression `\fresh(this)` dans la spécification d'un constructeur est une erreur, car c'est l'opérateur JAVA `new` qui effectue l'allocation mémoire de l'objet ; le constructeur se contente uniquement d'initialiser cet espace mémoire déjà alloué.