Sur les invariants de classes (Java) et de structures (C)

Sylvain Boulmé, Claude Marché

LSR-IMAG
INRIA Project ProVal & Université Paris Sud

ProVal
Laboratoire LRI

Contexte

- Difficultés de la prise en compte des invariants par Krakatoa
  - Invariants des objets accessibles
  - Source d'incorrection avec la ré-entrance
- Pas d'invariants associés aux structures en Caduceus

Invariants de classe : sémantique de JML

JML: « à l'appel d'une méthode, tous les invariants de tous les objets accessibles sont supposés valides »
Mais Krakatoa ne prend pas en compte tous les accessibles :

- seulement les arguments de type objet de la méthode
- pas les attribut de type objet d'un objet
- pas tableaux d'objets en arguments
- pas le résultat d'une méthode

Exemple 1

```
class A {
   ...
}

class B {
  A x;
  A m(B a,A b[]) {
     // a.x, b[0], b[1], etc.
     // et le résultat sont supposés vérifier
     // l'invariant de la classe A
     ...
  }
}
```

Exemple 2

```
class A {
      int x; // invariant x != 0;
}

class B {
      A i;
      void m() {
          this.i.x = 0;
      }
}
```

- Krakatoa dit que c'est correct alors que `m()` viole l'invariant de `this.i`
- Mais Krakatoa est correct vis-à-vis de sa propre sémantique !

Exemple 3

Structures chaînées : encore plus compliqué

```
class A {
      int x; //@ invariant x != 0;

      A suiv;

      int m() {
          A t = this;
          int z = 0;
          while (t !=null ) {
                z = z + 100 / t.x;
                t = t.suiv;
          }
      return z;
      }
}
```

On doit pouvoir prouver qu'il n'y a pas de division par zéro.

Exemple 4 : ré-entrance

```
abstract class A {
    /*@ normal_behavior
      @  assignable \nothing;
      @  ensures \result != 0;
      @*/
    abstract int m1();
}

class C {
    static int m2(A a) {
        return 100/a.m1();
    }
}
```

Jusque là tout va bien.

Exemple 4 : ré-entrance

```
class B extends A {
    int x; //@ invariant 0 < x;

    int m1() { return x; }

    int m3() {
        x = x-1;
        int z = C.m2(this);
        x = x+1;
        return z;
    }
}
```

Division par zéro non détectée...

→ impossibilité de raisonner de façon modulaire

Solution alternative : approche « Boogie »

Contexte

- C# : style Java/C++ « en plus propre »
- Spec# : langage de spécification pour C#
- Boogie : langage intermédiaire pour la vérification de programmes Spec#

Solution aux problèmes d'invariants de classe proposée dans: *Verification of object-oriented programs with invariants*. Barnett, DeLine, Fähndrich, Leino, Schulte. Microsoft Research.

Idée générale

- Ni invariant faible, ni invariant fort.
- Chaque objet a un champ implicite `inv` booléen, qui indique s'il vérifie son invariant.
- L'utilisateur contrôle les portions de programme où il s'autorise à temporairement violer un invariant: annotations `unpack(x)` et `pack(x)`

Exemple

```
class Purse {
      int intPart;
      int decPart;
      //@ invariant 0 <= decPart < 100;

      /*@ requires 0 <= cent < 100 */
      void credit(int euros, int cent) {
          //@ unpack(this);
          intPart += euros;
          decPart += cent;
          if (decPart >= 100) { decPart -= 100; intPart++ }
          //@ pack(this);
      }
}
```

pack et unpack

Signification donnée en pseudo-syntaxe Why :

```
unpack : (x:object) ->
  { x <> null and x.inv = true }
   x.inv ← false

pack : (x:object) ->
  { x <> null and x.inv = false and Inv_T(x) }
  x.inv ← true
```

où $Inv_T(x)$ est l'invariant du type T.

Résultat 1

Hypothèses :

- la mise à jour de champ $x.f = e$ est soumise à la précondition $x.inv = false$
- L'invariant $Inv_T(x)$ d'un objet x ne dépend que des champs de x.

Alors

$$\forall x : T,\ x.inv = true \rightarrow Inv_T(x)$$

est un invariant fort de tout programme.

Invariants dépendants des sous-objets

```
class A {
  int f;
  //@ requires this.inv = true
  void m() {
    //@ unpack(this);
    f = -1;
    ...
  }
}

class B {
  A g; //@ invariant this.g.f >= 0

  //@ requires this.inv = true
  void n() {
    g.m();
    ...
  }
}
```

Notion de composant

- Certains champ peuvent être déclarés comme *composant* de l'objet : modifieur `rep`.
- $Inv_T(x)$ peut dépendre de $x.f_0.\cdots.f_n.g$ si tous les f_i sont déclarés `rep`

Intuition :

- un objet est une boîte, le champ `inv` indique si elle est ouverte ou fermée.
- une boîte fermée doit satisfaire son invariant
- les champs `rep` sont à l'intérieur de la boîte
- une boite fermée ne contient que des boites fermées.

Pour chaque objet on introduit un champ booléen implicite `committed` qui signifie que cet objet est dans une boite fermée.

pack et unpack modifiés

```
unpack : (x:object) ->
  { x <> null and x.inv = true
     and x.committed = false}
  x.inv ← false;
  pour chaque f de Rep_T(x) :
     if x.f <> null then x.f.committed ← false

pack : (x:object) ->
  { x <> null and x.inv = false and Inv_T(x)
    and [ f in Rep_T(x) ] (x.f <> null -> x.f.inv = true) }
  x.inv ← true;
  pour chaque f de Rep_T(x) :
     if x.f <> null then x.f.committed ← true
```

Résultat 2

$$\forall x : T \quad (x.committed = true \rightarrow x.inv = true) \\ \wedge (x.inv = true \rightarrow (Inv_T(x) \wedge \\ \forall f \in Rep_T(x)(x.f \neq null \rightarrow x.f.committed = true)$$

est un invariant fort de tout programme.
Pour le montrer, il faut généraliser en ajoutant que chaque objet committed a un unique propriétaire.

Extension aux sous-classes

- Le champ `inv` n'est plus booléen, c'est un identificateur de classe.
- `x.inv = C` signifie que $Inv_D(x)$ est valide dès que C est une sous-classe de D.
- pack et unpack prennent en deuxième argument un nom de classe.
- $x.f = e$ est autorisé si x.inv est une superclasse stricte de la classe où f est déclarée.

Extension aux tableaux (suggestions)

Tableaux (suggestions) :

- un champ `committed` pour le tableau
- l'affectation $t[i] = e$ est soumise à la précondition $t.committed = false$
- pour les tableaux d'objets : un champ `inv` pour chaque élément (donc en fait rien à ajouter !)

Exemple :

```
class IntSet {
   /*@ rep @*/ int contents[]
   //* invariant \forall int i;
              0 < i < t.length ; t[i-1] <= t[i]
```

toute modification d'un `contents[i]` est soumise à `t.committed = false` donc un `unpack` avant.

Application au langage C (suggestions)

- Structures (analogie avec objets Java) : champ booléen `inv`
- Arithmétique de pointeurs : champ `committed` sur l'adresse de base d'un pointeur.
- Union (analogie avec sous-classes) : nouveau champ indiquant quelle est le cas de l'union actuellement valide.

Exemple :

```
typedef struct queue {
  /*@ rep @*/ char contents[];
  int length;
  int first, last;
  unsigned int empty, full :1;
} queue;
/*@ invariant queue q;
  @   \valid_range(q.contents, 0, q.length-1) &&
  @   0 <= q.first < q.length &&
  @   0 <= q.last < q.length
  @*/
```

Conclusion

- méthodologie garantie correcte, aisément implantable.
- permet à l'utilisateur de contrôler quand il veut qu'un invariant soit vrai.
- mais: demande une insertion manuelle d'annotations
- perspective : génération automatique de ces annotations ?
  - annotation `rep` : calculable
  - idée : la plupart du temps, les invariants doivent être vrais $\longrightarrow$ ajout des unpack par nécessité $+$ pack correspondant en fin de méthode (peut être compliqué...)