TD6 – preuve de programmes avec Why3

L'objectif de ce TP est de spécifier et prouver des programmes écrits en Java avec Krakatoa [1]. Les programmes sont d'abord annotés en utilisant le langage de spécification *JML* (*Java Modeling Language*) sous la forme de pré et post-conditions, et d'invariants. Krakatoa est un *front-end* de l'environnement de preuve Why3 [2] qui traduit le code annoté dans un langage fonctionnel appelé WhyML. L'exécution de Why3 génère un ensemble de conditions à prouver (appelées obligations de preuve), exprimées sous la forme de triplets de Hoare. Les règles d'inférence du calcul de Hoare sont ensuite appliquées à ces conditions. Enfin, un prouveur automatique ou interactif, tel que Alt-Ergo, est appelé sur les formules de logique classiques restantes (provenant de l'application de la règle de conséquence par exemple).

Installation de Why3 et de Alt-Ergo

1. Rendez-vous sur la page `http://doc.eisti.fr/linux/#install-why-why3-ubuntu` et suivez les instructions pour installer Why3. Les outils Alt-Ergo et Krakatoa sont également automatiquement installés.
2. Afin que Why3 détecte les prouveurs installés (ici, Alt-Ergo), exécutez les commandes suivantes :

   ```
   $ why3 config --detect
   $ why3 --list-provers
   ```

   Si tout va bien, vous devriez voir :

   ```
   Known provers:
     Alt-Ergo (0.95.2)
     ...
   ```

Syntaxe JML

Les commentaires spéciaux contenant du JML sont de la forme : `/*@ <code JML> @*/` ou `//@ <code JML jusqu'en fin de ligne>`. Vous ne pouvez mettre qu'un seul prédicat ou un seul lemme dans chaque commentaire spécial. Les mots clé suivants peuvent être utilisés :

requires pour donner une pré-condition :

```
requires n >=0 ;
```

ensures pour donner une post-condition :

```
ensures \result == n*n ;
```

loop_invariant pour donner l'invariant d'une boucle :

```
loop_invariant i+j == n ;
```

1. `http://krakatoa.lri.fr/`
2. `http://why3.lri.fr/`

loop_variant pour donner un variant d'une boucle (valeur qui décroit à chaque tour de boucle et reste positive ou nulle) :

```
loop_variant j - i ;
```

lemma pour donner un lemme (permet de guider le prouveur) :

```
lemma mul_distr_r: \forall integer a b c; (a+b)*c == a*c + b*c ;
```

assert pour indiquer qu'une propriété est vraie à ce point du programme :

```
assert i > 0 && j > 0 ;
```

predicate pour définir un prédicat logique :

```
predicate even(integer n) = n%2 == 0 ;
```

result pour faire référence à la valeur de retour d'une fonction

old pour faire référence à la valeur initiale d'une variable dans la post-condition :

```
\old(i)
```

forall pour le quantificateur universel :

```
\forall integer n; n+1 > n;
```

exists pour le quantificateur d'existence :

```
\exists integer k; 4 == 2*k ;
```

==> pour l'implication :

```
\forall integer x; x == 3 ==> x/2 == 1 ;
```

<==> pour l'équivalence :

```
\forall integer x; x%2 == 0 <==> x == 2*(x/2) ;
```

ghost pour introduire des variables fantômes qui instrumentent le code :

```
ghost integer gx = x ;
```

Exercice 1

Considérons une méthode statique `max` qui calcule le maximum de deux entiers.

```
//@+ CheckArithOverflow = no
public class TPWhy {
  /*@ ensures
    @    \result >= x && \result >= y ;
    @*/
  public static int max(int x, int y) {
    if (x>y)
      return x;
    else
      return y;
  }
}
```

Le commentaire JML juste avant la méthode `max` est un contrat spécifiant un comportement prévu pour `max`. La spécification d'un programme est donnée par une pré-condition (introduite par le mot clé `requires`) et une post-condition (introduite par `ensures`). Le mot clé `\result` représente le résultat de la fonction. La formule de la post-condition de `max` signifie que le valeur retournée est supérieure ou égale à `x` et à `y`. Afin d'ignorer tout débordement arithmétique (*e.g.*, la valeur d'une variable entière dépasse la taille du type `int`), le commentaire JML `//@+ CheckArithOverflow = no` est ajouté au début du fichier (plutôt que de rajouter une pré-condition sur la valeur maximale de la variable). Pour l'instant, la question ne se pose pas.

1. Exécutez la commande suivante : `krakatoa TPWhy.java`

   Cette commande lit le fichier donné et lance le traducteur de Java vers WhyML puis en logique de Hoare sur le programme annoté, ainsi qu'une interface graphique. Les formules logiques générées sont appelées des obligations de preuve. Elles expriment la validité du programme. La dernière phase (*i.e.*, la vérification des formules) n'a pas encore eu lieu. La fenêtre qui apparaît est découpée en trois parties :
   - à gauche se trouve une barre d'outils ;
   - au centre figure l'arborescence de toutes les obligations de preuves du programme, c'est-à-dire les propriétés à prouver pour montrer la correction du programme par rapport à sa spécification ;
     - Il y a 4 obligations de preuve générées pour ce programme, données dans le dossier `Jessie_program`
     - Généralement, il y a une obligation de preuve pour chaque comportement de chaque méthode ou constructeur.
     - La sûreté (*safety*) est un de ces comportements qui comprend la vérification de l'absence d'erreurs à l'exécution, telles que la division par zéro, le déréférencement d'un pointeur `null`, l'accès à un tableau hors de ses bornes, *etc.*
   - à droite dans l'onglet `Source code` se trouve le programme à prouver ;
   - à droite dans l'onglet `Task` apparaît l'obligation de preuve sous la forme d'une formule logique : la liste des hypothèses et la conclusion à prouver introduite par le mot clé `goal`.

   Lorsqu'on sélectionne l'obligation de preuve du programme et qu'on clique sur `Split` dans le menu de gauche, Why3 découpe l'obligation de preuve en un ensemble de sous-buts. On peut voir l'instruction du code et la formule logique correspondant à une obligation de preuve en cliquant dessus.
2. Sélectionnez le dossier `Jessie_program` et lancez Alt-Ergo en cliquant sur le bouton correspondant dans le menu de gauche. L'outil réussit à prouver toutes les propriétés demandées. Peut-on déduire que le programme est correct ?
3. Changez la spécification pour capturer le fait que le résultat doit être le maximum de x et y en ajoutant `&& \forall integer z; z >= x && z >= y ==> z >= \result` (signifiant que le résultat est le plus petit entier à la fois plus grand que `x` et `y`) dans la post-condition, et lancez de nouveau le prouveur.
4. Remplacez l'instruction `return y` par l'instruction `return x` et lancez de nouveau le prouveur. La post-condition de `max` n'est plus prouvée. Puisque c'est une conjonction de plusieurs propositions, c'est une bonne idée de découper l'obligation de preuve en un ensemble de sous-buts.
5. Sélectionnez l'obligation de preuve et cliquez sur le bouton `Split`. Il en résulte un sous-but. Lancez de nouveau le prouveur pour vérifier ce sous-but. Il reste non prouvé. Il peut être découpé en trois sous-buts. Lancez encore une fois le prouveur sur chacun d'eux. Le deuxième sous-but reste non prouvé. La spécification correspondante est montrée dans l'onglet `Source code` : c'est la partie `\result >= y` qui n'est pas valide. En effet, le résultat n'est pas nécessairement supérieur ou égal à `y` dans la seconde branche.
6. Corrigez la faute introduite dans le code et lancez de nouveau le prouveur.

Exercice 2

1. Soit la méthode statique `findMax` qui trouve le maximum d'un tableau d'entier :

```
public static int findMax(int[] t) {
  int m = t[0];
  int r = 0;
```

```
  for (int i=1; i < t.length; i++) {
    if (t[i] > m) {
      r = i;
      m = t[i];
    }
  }
  return r;
}
```

(a) Annotez le programme par sa spécification sous forme de pré et post-conditions.

(b) Ajoutez l'invariant nécessaire pour prouver la boucle `for`, ainsi que le variant pour prouver la terminaison.

2. Soitz la méthode statique `allZeros` qui retourne `true` si les valeurs de toutes les cases d'un tableau d'entiers sont nulles et `false` sinon :

```
public static boolean allZeros(int[] t) {
  boolean r = true;
  int i = 0;
  while (i < t.length && t[i] == 0) {
    i = i + 1;
  }
  if (i != t.length) {
    r = false;
  }
  return r;
}
```

(a) Même questions que précédemment.

3. Même questions que précédemment avec les programmes du TD précédent.