Karlsruher Institut für Technologie
Institut für Theoretische Informatik
Prof. Dr. Bernhard Beckert
Thorsten Bormer, Dr. Vladimir Klebanov, Dr. Mattias Ulbrich

Formale Systeme, WS 2014/2015

Übungsblatt 10

Dieses Übungsblatt wird in der Übung am 23.01.2015 besprochen.

Wir haben unter Adresse `http://formal.iti.kit.edu/teaching/FormSysWS1415/jmlcheck` einen Syntaxchecker für JML installiert. Sie können dort für Ihre Lösungen überprüfen, ob sie syntaktisch korrekt sind.

Aufgabe 1

Zur Implementierung einer Personendatenbank wird folgende Java-Klasse verwendet:

```
class Person {
   String name;
   int age;
   boolean isFemale;
   Person father;
   Person mother;
   Person[] children;

   void celebrateBirthday() {
      age ++;
   }

   void becomeParentTo(Person child) {
      children = addToArray(child);
      if(isFemale) {
         child.mother = this;
      } else {
         child.father = this;
      }
   }

   Person[] addToArray(Person child) {
      Person[] result = new Person[children.length + 1];
      System.arraycopy(children, 0, result, 0, children.length);
      result[children.length] = child;
      return result;
   }
}
```

(a) Formulieren Sie für die folgenden Aussagen Klassen-Invarianten in JML:

(i) *Der Vater einer Person ist männlich und älter als die Person selbst.*

(ii) *Alle Kinder einer Person sind echt jünger als die Person selbst.*

(iii) *Jede Person ist Kind ihrer Mutter.*

(iv) *Wenn eine Person weiblich ist, dann ist sie Mutter all ihrer Kinder.*

(b) Formulieren Sie für die Methode `celebrateBirthday()` einen Methoden-Vertrag in JML, der folgendes besagt: *Die Methode darf in jedem Zustand aufgerufen werden. Nach Ausführung der Methode ist der Wert des Feldes* `age` *um genau 1 erhöht, alle anderen Speicherstellen sind unberührt geblieben.*

(c) Beschreiben Sie das Verhalten der Methode `becomeParentTo(Person)` möglichst präzise mit einem JML-Vertrag.

(d) Vervollständigen Sie folgenden Methodenvertrag für die Methode `addToArray(Person)`:

```
/*@ public normal_behaviour
  @  ensures \result.length == ...
  @  ensures (\forall int i; 0 <= i && ...
  @  assignable \nothing;
  @*/
```

(e) Die JML-Semantik gestattet es nicht, dass eine Person ohne Vater/Mutter angelegt wird. Wie muss die Spezifikation geändert werden, wenn Personen ohne Angabe ihrer Eltern angelegt werden können sollen?

Lösung zu Aufgabe 1

(a) Die folgenden Klasseninvarianten können an Stellen in der Klassendefinition von `Person` stehen, an der Felder oder Methoden deklariert werden können.

Invarianten gelten für das durch `this` referenzierte Objekt. Die Aussage ist implizit allquantifiziert, da wir annehmen, dass Invarianten für *alle* Objekte gelten. Es genügt also für Aussagen die Eigenschaften für "alle Personen" fordern, diese für das durch `this` referenzierte Objekt zu formalisieren.

(i)
```
//@ invariant !father.isFemale && father.age > this.age;
```

(ii)
```
//@ invariant (\forall int i; 0<=i && i<children.length; children[i].age<age);
```

(iii)
```
/*@ invariant (\exists int i; 0<=i && i<mother.children.length;
  @                mother.children[i] == this);
  @*/
```

(iv)
```
/*@ invariant isFemale ==>
  @     (\forall int i; 0<=i && i<children.length;
  @          children[i].mother == this);
  @*/
```

(b) Die Forderung, dass die Methode "in jedem Zustand aufgerufen werden" darf, beschreiben wir in JML durch die allgemeingültige Vorbedingung `true`[1].

[1] Wird keine Vorbedingung angegeben, so wird als Standard `true` verwendet, so dass wir in diesem Fall auch auf eine Vorbedingung ganz verzichten hätten können.

```
/*@ public normal_behaviour
  @   requires true;
  @   ensures age == \old(age) + 1;
  @   assignable age;
  @*/
void celebrateBirthday() { ... }
```

Nebenbemerkung: Nach der Definition von "Invariante" müssen alle Operationen alle Invarianten erhalten. Dies ist für diese Methodendefintion nicht gewährleistet! Betrachten wir z. B. die folgende Situation mit zwei Personen-Objekten `p` und `q`, für die gilt, dass `p.age == 20`, `q.age == 21` und `p.father == q`. Die weiteren Felder seien so gewählt, dass die Invarianten alle erfüllt sind.

Nach dem Methodenaufruf `p.celebrateBirthday()` gilt `p.age == 21` und damit ist die Invariante aus (a.i) *nicht* mehr erfüllt.

Damit die Methode `celebrateBirthday` also die Invarianten erhält, müsste eigentlich die stärkere Vorbedingung `age < father.age-1 && age < mother.age-1;` gefordert werden.

(c) Folgender Vertrag beschreibt das Verhalten der Methode möglichst präzise:

```
/*@ public normal_behaviour
  @   requires child.age < this.age;                                    (1)
  @   ensures child.mother == isFemale ? this : \old(child.mother);     (2)
  @   ensures child.father == isFemale ? \old(child.father) : this;     (3)
  @   ensures children.length == \old(children.length) + 1;             (4)
  @   ensures (\forall int i; 0<=i && i<children.length-1;              (5)
  @                children[i] == \old(children[i]));
  @   ensures children[children.length-1] == child;                     (6)
  @   assignable children, child.father, child.mother;                  (7)
  @*/
void becomeParentTo(Person child) { ... }
```

Einige Beobachtungen dazu:

- Die Vorbedingung (1) ist, dass das Alter des Kindes unter dem dieser Person liegt, damit die Invariant aus (a.i) nicht verletzt wird.
- Bei (2) und (3) fällt auf, dass nicht nur beschrieben werden muss, wie sich `child.mother` and `child.father` *ändern*, sondern auch unter welchen Bedingungen sie sich *nicht ändern*.
- Für das Feld `this.children` muss sowohl die Änderung der Länge beschrieben werden (4), also auch der neue Inhalt (5), (6).
- Die Methode ändert (7) die Referenz auf `children` und die Felder `father` und `mother` für die Referenz `child`. Beachten Sie: Der Inhalt des *alten* Feldes `children`, also die Speicherstellen, die durch `children[*]` bezeichnet werden, werden von dieser Implementierung in keinem Fall verändert.

(d)
```
/*@ public normal_behaviour
  @  ensures \result.length == children.length + 1;
  @  ensures (\forall int i; 0 <= i && i < children.length;
  @               \result[i] == children[i]);
  @  ensures \result[children.length] == child;
  @  assignable \nothing;
  @*/
```

Die `assignable`-Klausel mag im ersten Moment ungewöhnlich erscheinen: Wie kann die Methode nichts verändern, wenn sie doch ein neues Objekt auf dem Speicher anlegt? Antwort: Assignable-Klauseln beziehen sich auf den schon existierenden Teil des Speichers. Die Methode darf neuen Speicher belegen, aber *nichts* an den schon existierenden Speicherstellen manipulieren.

Alle anderen Teile des Systems hängen nur von schon existierenden Speicherstellen ab, daher kann die Methode das System nicht "verändern".

(e) JML hat eine Standardeinstellung: Wenn die Deklaration eines Feldes, eines Methodenparameters oder eines Rückgabewertes nicht weiter annotiert ist, so gilt er als `nonnull`: An der entsprechenden Stelle im Speicher darf der Wert `null` *nie* stehen.

Um diesen Standard zu deaktivieren, muss das Schlüsselwort `nullable` als Modifikator verwendet werden:

```
/*@ nullable */ Person father;
/*@ nullable */ Person mother;
```

Mit dieser modifizierten Deklaration, sollten nur auch die Spezifikationen angepasst werden und den weiteren Fall in Betracht ziehen. So sollte z.B. die Invariante aus (a.i) folgendermaßen lauten:

```
//@ invariant father!=null ==> (!father.isFemale && father.age > this.age);
```

Aufgabe 2

(a) Schreiben Sie eine Java-Methode, die die folgende Spezifikation erfüllt: Gegeben ein Feld[2] a von ganzzahligen Werten $a_1, \ldots, a_n$ soll die Methode Index i_0 des ersten Auftretens der Zahl 0 liefern, also den Index i_o ausgeben, so dass

(i) $a_{i_0} = 0$ und

(ii) $a_k \neq 0$ für alle $0 \leq k < i_0$.

Falls keiner der Werte $a_1, \ldots, a_n$ gleich 0 ist, so soll die Methode -1 zurückliefern.

(b) Schreiben Sie einen JML-Methodenvertrag, der diese Spezifikation wiedergibt.

(c) Die Methodenimplementierung wird eine Schleife benutzen. Geben Sie für die Schleife eine möglichst starke Invariante und eine Variante[3] an.

Lösung zu Aufgabe 2

Das folgende annotierte Programm beinhaltet die Lösung für alle drei Teilaufgaben. Beachten Sie in der Invariante, dass dort `k <= array.length` steht, auch wenn für den letzten Schleifendurchlauf die Laufvariable `k` den Wert `array.length-1` annimmt.

Die Schleifeninvariante muss aber auch *nach* dem letzten Schleifendurchlauf noch gelten, daher muss in der Schleifeninvariante der größere Bereich gewählt werden.

[2] engl. array

[3] engl. decreases clause

```
public class ZeroFinder {

    /*@ public normal_behaviour
      @   requires true;
      @   ensures -1 <= \result && \result < array.length;
      @   ensures \result >= 0 ==> array[\result] == 0 &&
      @              (\forall int i; 0<=i && i<\result; array[i] != 0);
      @   ensures \result == -1 ==>
      @              (\forall int i; 0<=i && i<array.length; array[i] != 0);
      @   assignable \nothing;
      @*/
    public int findZero(int[] array) {
        int k = 0;

        /*@ loop_invariant
          @   0 <= k && k <= array.length &&
          @   (\forall int i; 0<=i && i<k; array[i] != 0);
          @ decreases array.length - k;
          @*/
        while(k < array.length) {
            if(array[k] == 0) {
                return k;
            }
            k++;
        }

        return -1;
    }
}
```

Aufgabe 3

Für eine Anwendung im Bankenbereich wird die folgende Klasse zur Modellierung von Geldbeträgen verwendet:

```
class Betrag {
   int euros;
   int cents;

   Betrag(int euros, int cents) {
      this.euros = euros;
      this.cents = cents;
   }

   Betrag add(Betrag b) { ... }
}
```

Es ist das Ziel, die Benutzung dieser Klasse auf *normalisierter Beträge* einzuschränken. Ein Betrag heißt normalisiert, wenn er nicht-negativ ist und der Cent-Anteil im Betrag weniger als einen Euro ausmachen. Die Einschränkung in der Verwendung der Klasse soll mittels Annotatationen in JML formal umgesetzt werden.

(a) Geben Sie eine JML-Klasseninvariante für die Klasse `Betrag` an, die besagt, dass das Betrag-Objekt normalisiert ist.

(b) Geben Sie einen Vertrag für die Method `add` an, der besagt, dass der Betrag, der als Ergebnis zurückgeliefert wird, der Summe des Arguments und des Aufrufempfängers entspricht.

Die Methode darf ein neues Ergebnisobjekt erstellen, darf aber keine existierenden Speicherstellen abändern.

(c) In einem ersten Versuch wird die Methode `add` nun folgendermaßen implementiert:

```
Betrag add(Betrag b) {
   int e = euros + b.euros;
   int c = cents + b.cents;
   return new Betrag(e, c);
}
```

Warum verstößt diese Implementierung gegen die JML-Spezifikation und wie kann das repariert werden?

Lösung zu Aufgabe 3

(a) Die Normalisierung kann direkt als 3 Ungleichungen formuliert werden, die die Bereiche für die Felder `euros` and `cents` einschränken.

```
/*@ invariant 0 <= euros && 0 <= cents && cents < 100;  */
```

(b) Der "Wert" eines Betrages ist `100*euros + cents`. Im Vertrag muss also gefordert werden, dass der Wert des Ergebnisses gerade die Summe aus dem Wert von `b` und `this` ist. Der Vertrag lautet also:

```
/*@ ensures \result.euros*100 + \result.cents ==
  @    this.euros*100 + this.cents + a.euros*100 + a.cents;
  @ assignable \nothing;
  @*/
```

Die `assignable`-clause `\nothing` erlaubt, dass neue Objekte erstellt werden und deren Felder geändert werden.

(c) Das Problem ist, dass nach der Ausführung des Konstruktors am Ende der Methode `add` die Invariante des neu erzeugten Objektes unter Umständen verletzt ist und die Spezifikation damit nicht erfüllt wird. Die Klasseninvariante muss schließlich von jedem Konstruktor etabliert werden.

Um dem Problem zu entgehen, müssen die Argumente für den Konstruktor normalisiert werden. Dies kann z. B. erreicht werden, indem die zusätzliche Anweisung

```
if(c >= 100) {
   c -= 100;
   e ++;
}
```

direkt vor der `return`-Anweisung der Methode `add` eingebaut wird.