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

Formale Systeme, WS 2015/2016

Lösungen zu Übungsblatt 10

Dieses Übungsblatt wurde in der Übung am 22.01.2016 besprochen.

Aufgabe 1

Zur Implementierung einer Personendatenbank wird folgende Java-Klasse verwendet:

```
class Person {
   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 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.*

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

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

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

Lösung zu Aufgabe 1

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

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

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

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

Einige Beobachtungen dazu:

- Bei (1) und (2) 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*.

[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.

- Für das Feld `this.children` muss sowohl die Änderung der Länge beschrieben werden (3), also auch der neue Inhalt (4), (5).
- Die Methode ändert (6) 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.

(c)
```
/*@ 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".

Aufgabe 2

(a) Schreiben Sie eine Java-Methode, die die folgende Spezifikation erfüllt: Gegeben ein Array 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.

Lösung zu Aufgabe 2

Das folgende annotierte Programm beinhaltet die Lösung für beide Teilaufgaben.

```
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;

        while(k < array.length) {
            if(array[k] == 0) {
                return k;
            }
            k++;
        }

        return -1;
    }
}
```