Tentamen Formele Methoden voor Software Engineering (213520)

16 april 2009, 9.00-12.30 uur.

BELANGRIJK: geef op je tentamen duidelijk aan:

- je studierichting
- of je beide huiswerkopgaven gemaakt hebt, en bij welke werkcollegeleider.

Bij dit tentamen mag je de de FSP Quick Reference Card en de JML Cheat Sheet gebruiken, alsmede de sheets van de colleges (geen boeken).

---

1. *(25 punten)*

   In een zonnebankcentrum kan een klant een solarium cabine reserveren bij de toonbank. Zodra een cabine vrijkomt wordt ie aangezet, en wordt het nummer van de cabine aan de klant gegeven. De klant gaat naar de aangegeven cabine en geeft daar aan hoeveel tijd de cabine in werking moet zijn. Nadat de cabine stopt signaleert de cabine naar de toonbank het bedrag dat de klant moet betalen. De klant betaalt vervolgens bij de toonbank (ga er niet vanuit dat de klant precies weet wat ie moet betalen). Verder kan de desk een cabine laten schoonmaken (uiteraard als er geen klant in de cabine is).

   (a) Specificeer het zonnebankcentrum voor 1 klant en 1 cabine. Neem het volgende proces voor de toonbank:

   ```
   DESK = ( reserve -> turn_on -> get_number -> DESK
          | amount[m:M] -> pay[m] -> DESK
          | clean -> DESK).
   ```

   (hint: specifeer processen CUSTOMER en CABIN).

   (b) Specificeer nu een zonnebankcentrum met 2 cabines en 3 klanten (hint: denk goed na over hoe je het proces DESK moet veranderen om deadlocks te voorkomen).

   (c) Specificeer een safety property die zegt dat het verschil tussen het aantal `reserve` acties en het aantal `pay` acties nooit groter is dan 2. Geldt deze eigenschap voor jouw zonnebankcentrum?

   (d) Specificeer een progress property die zegt dat klant nummer 3 uiteindelijk een cabine krijgt. Geldt deze eigenschap? Geeft nu prioriteit aan de `reserve` acties van klant 1 en klant 2. Geldt de eigenschap nu nog steeds?

   (e) Voeg een proces toe aan het systeem dat ervoor zorgt dat elke cabine na precies 2 keer gebruikt te zijn wordt schoongemaakt.

(a) *(5 punten)*

```
const MONEY = 3
range M = 1..MONEY

DESK = ( reserve -> turn_on -> get_number -> DESK
       | amount[m:M] -> pay[m] -> DESK
       | clean -> DESK).

CABIN = (turn_on -> time[t:M] -> amount[t] -> CABIN
       | clean -> CABIN).

CUSTOMER = (reserve -> get_number -> time[t:M] -> pay[m:M] -> CUSTOMER).

||SOLARIUM = (DESK || CABIN || CUSTOMER)\{turn_on, amount[M]}.
```

(b) *(5 punten)*

```
const MONEY = 3
range M = 1..MONEY
const N_CUST = 3
const N_CAB = 2
range C = 1..N_CUST
range A = 1.. N_CAB

DESK1 = ( c[i:C].reserve -> a[j:A].turn_on -> c[i].get_number[j] ->
DESK1).

DESK2 =  ( a[j:A].amount[m:M] -> c[C].pay[j][m] -> DESK2
            | a[j:A].clean -> DESK2).

CABIN = (turn_on -> c[i:C].time[t:M] -> amount[t] -> CABIN
         | clean -> CABIN).


||CABINS = a[A]:CABIN.

CUSTOMER = (reserve -> get_number[j:A] -> a[j].time[t:M] -> pay[j][m:M] ->
CUSTOMER).

||CUSTOMERS = c[C]:CUSTOMER.


||SOLARIUM = (DESK1 || DESK2 || CABINS || CUSTOMERS)
/{forall[i:C][j:A][t:M] {c[i].a[j].time[t]/a[j].c[i].time[t]}}.
```

(c) *(5 punten)*

```
property REQUESTS = (c[C].reserve -> REQ[1]),
REQ[k:-3..3] = ( when(k<2) c[C].reserve -> REQ[k+1]
                | c[C].pay[A][M] -> REQ[k-1]).


||SOLARIUM = (DESK1 || DESK2 || CABINS || CUSTOMERS|| REQUESTS)
/{forall[i:C][j:A][t:M] {c[i].a[j].time[t]/a[j].c[i].time[t]}}.
```

De uitvoer van LTSA:

```
Trace to property violation in REQUESTS:
c.1.reserve
a.1.turn_on
c.1.get_number.1
c.2.reserve
a.2.turn_on
c.2.get_number.2
c.3.reserve
```

(d) *(5 punten)* De progress-eigenschap:

```
progress CUSTOMER3 = {c[3].a[A].time[M]}
```

This property holds, even if we give priority to the requests of customer 1 and 2, in the following way:

```
||SOLARIUM = (DESK1 || DESK2 || CABINS || CUSTOMERS)
/{forall[i:C][j:A][t:M] {c[i].a[j].time[t]/a[j].c[i].time[t]}}
<<{c[1].reserve. c[2].reserve}.
```

(e) *(5 punten)*

```
CLEANCHECK = CLEAN[0][0],
CLEAN[i:CL][j:CL] = (  when (i<N_CL) a[1].turn_on -> CLEAN[i+1][j]
                     | when (j<N_CL) a[2].turn_on -> CLEAN[i][j+1]
                     | when (i == N_CL) a[1].clean -> CLEAN[0][j]
                     | when (j == N_CL) a[2].clean -> CLEAN[i][0]).


||SOLARIUM = (DESK1 || DESK2 || CABINS || CUSTOMERS || CLEANCHECK)
/{forall[i:C][j:A][t:M] {c[i].a[j].time[t]/a[j].c[i].time[t]}}.
```

2. *(10 punten)* Beschouw de volgende methode:

```
int[] keerOm(int[] a) {
    int[] result = new int[a.length];
    int i = a.length-1;
    while (i >= 0) {
        result[a.length-1-i] = a[i];
        i--;
    }
    return result;
}
```

   (a) Specificeer een zo volledig mogelijk contract voor deze methode in JML.

   (b) Bewijs (met behulp van een lus-invariant) de correctheid van de methode. Geef de bewijsstappen duidelijk aan!

(Beoordeling: Geef max. 2 punten voor het contract, mex. 2 voor de invariant, en max. 6 voor de verschillende ingrediënten van het bewijs.)

(a) Het contract:

```
/*@
  @ requires a != null;
  @ ensures \result != null && \result.length == a.length;
  @ ensures (\forall int k; 0 <= k && k < a.length;
  @    \result[a.length-1-k] == a[k]);
  @*/
```

(b) De lusinvariant:

```
/*@ loop_invariant result.length == a.length && i>=-1;
  @ loop_invariant (\forall int k; i < k && k < a.length;
  @    result[a.length-1-k] == a[k]);
  @*/
while (i >= 0) {
    result[a.length-1-i] = a[i];
    i--;
}
```

We korten af:

$$\begin{aligned} R &= a \neq \mathsf{null} \\ E &= r \neq \mathsf{null} \wedge |r| = |a| \wedge \forall 0 \leq k < |a| : r[|a| - 1 - k] = a[k] \\ I &= |r| = |a| \wedge i \geq -1 \wedge \forall i < k < |a| : r[|a| - 1 - k] = a[k] \end{aligned}$$

Het bewijs valt uiteen in drie delen:

- $\{R\}$ `r=new int[a.length]; i=a.length-1;` $\{I\}$ (preconditie garandeert lusinvariant)

$$\begin{array}{ll} & wp(\texttt{r=new int[a.length]; i=a.length-1;}) \\ & \quad \{|r| = |a| \wedge i \geq -1 \wedge \forall i < k < |a| : r[|a| - 1 - k] = a[k]\} \\ = & \quad wp(\texttt{r=new int[a.length];}) \\ & \qquad \{|r| = |a| \wedge |a| \geq 0 \wedge \forall |a| - 1 < k < |a| : r[|a| - 1 - k] = a[k]\} \\ = & \quad |a| = |a| \\ = & \quad a \neq \mathsf{null} \end{array}$$

  Note that the introduction of the condition $a \neq \mathsf{null}$ in the last step is necessary due to the fact that any condition with $|a|$ implicitly implies $a \neq \mathsf{null}$, but this implicit condition disappears when we replace $|a| = |a|$ with true.
- $\{I \wedge i \geq 0\}$ `r[a.length-1-i]=a[i]; i--;` $\{I\}$ (lusinvariant is correct)

  Hievoor moeten we bewijzen dat de preconditie de zwakste preconditie impliceert:

$$\begin{array}{ll} & wp(\texttt{r[a.length-1-i]=a[i]; i--;}) \\ & \quad \{|r| = |a| \wedge i \geq -1 \wedge \forall i < k < |a| : r[|a| - 1 - k] = a[k]\} \\ = & \quad wp(\texttt{r[a.length-1-i]=a[i];}) \\ & \qquad \{|r| = |a| \wedge i \geq 0 \wedge \forall i \leq k < |a| : r[|a| - 1 - k] = a[k]\} \\ = & \quad |r| = |a| \wedge i \geq 0 \wedge a[i] = a[i] \wedge \forall i < k < |a| : r[|a| - 1 - k] = a[k] \\ = & \quad |r| = |a| \wedge i \geq 0 \wedge \forall i < k < |a| : r[|a| - 1 - k] = a[k] \end{array}$$

  Aangezien duidelijk is dat

$$\begin{array}{rl} & |r| = |a| \wedge i \geq -1 \wedge \forall i < k < |a| : r[|a| - 1 - k] = a[k] \wedge i \geq 0 \\ = & \quad |r| = |a| \wedge i \geq 0 \wedge \forall i < k < |a| : r[|a| - 1 - k] = a[k] \end{array}$$

  zijn we klaar.
- $I \wedge i < 0 \Rightarrow E$ (lusinvariant garandeert postconditie)

$$\begin{array}{rl} & |r| = |a| \wedge i \geq -1 \wedge \forall i < k < |a| : r[|a| - 1 - k] = a[k] \wedge i < 0 \\ = & \quad |r| = |a| \wedge i = -1 \wedge \forall i < k < |a| : r[|a| - 1 - k] = a[k] \\ \Rightarrow & \quad r \neq \mathsf{null} \wedge |r| = |a| \wedge \forall 0 \leq k < |a| : r[|a| - 1 - k] = a[k] \end{array}$$

3. *(15 punten)* Terug naar de sauna. Beschouw de volgende declaraties:

```
interface Cabin {
    /** Sets the customer field to a given value. */
    void setCust(Customer cust);
    /** Returns the current value of the customer field. */
    Customer getCust();
    /** Returns true if the customer is currently null. */
    boolean isFree();
    /** Returns the rate of this cabin (a positive amount). */
    double getRate();
}

class Customer {
    /** Returns the current payable amount. */
    double getAmount() {
        return amount;
    }
    /** Decreases the payable amount by a (positive) payment. */
    void payAmount(double payment) {
        amount -= payment;
    }
    /**
     * Reserves a cabin (which should be empty at the time of call)
     * and sets the payable amount to the cabin rate.
     * Should only be called if the customer has no cabin yet.
     */
    void setCabin(Cabin cabin) {
        cabin.setCust(this);
        this.cabin = cabin;
        this.amount = cabin.getRate();
    }
    /**
     * Frees the current cabin of this customer. Should only be called
     * if the customer has a cabin, and the due amount has been payed.
     */
    void resetCabin() {
        this.cabin.setCust(null);
        this.cabin = null;
    }
    /** Returns the currently reserved (possibly null) cabin. */
    Cabin getCabin() {
        return cabin;
    }
    // The current payable amount; should be non-negative.
    private double amount = 0;
    // The currently reserved cabin. If null, the payable amount
    // should be zero.
    private Cabin cabin;
}
```

(a) Stel een contract in JML op voor `Cabin`, waarin het JavaDoc-commentaar zoveel mogelijk formeel gespecificeerd is.

(b) `Cabin` wordt geïmplementeerd in een klasse `CabinImpl`. Geef van `CabinImpl` de constructor (waarin de `rate` wordt gezet) en de instantievariabelen, met bijbehorend JML-commentaar zodat de specificatie correct te bewijzen is. (Het bewijs zelf hoeft niet geleverd te worden!)

(c) Stel een contract in JML op voor `Customer`, waarin het JavaDoc-commentaar zoveel mogelijk formeel gespecificeerd is.

(d) Bewijs de correctheid van de methode `resetCabin()`.

(a) *(3 punten)* Het contract van `Cabin`:

```
interface Cabin {
    //@ ensures this.cust == cust;
    void setCust(Customer cust);
    //@ ensures \result == cust;
    //@ pure
    Customer getCust();
    //@ ensures \result == (cust == null);
    //@ pure
    boolean isFree();
    //@ ensures \result>0;
    //@ pure
    double getRate();
    //@ instance model Customer cust;
}
```

(b) *(2 punten)* De gevraagde declaraties:

```
    //@ requires rate > 0;
    public CabinImpl(double rate) {
        this._rate = rate;
    }

    private final double _rate;
    private Customer _cust;
    //@ private represents cust = _cust;
    //@ private invariant _rate > 0;
```

(c) *(5 punten)* Het contract van `Customer`:

```
    //@ ensures \result == amount;
    double getAmount() {
        ...
    //@ requires cabin != null;
    //@ requires payment > 0 && payment <= amount;
    //@ ensures amount == \old(amount)-payment;
    void payAmount(double payment) {
        ...
    /*@ requires this.cabin == null && cabin != null && cabin.isFree();
      @ ensures this.cabin == cabin && cabin.getCust() == this;
      @ ensures this.amount == cabin.getRate();
      @*/
    void setCabin(Cabin cabin) {
        ...
    /*@ requires cabin != null && amount == 0;
      @ ensures cabin == null && \old(cabin).isFree();
      @*/
    void resetCabin() {
        ...
    /*@ ensures \result == cabin; */
    Cabin getCabin() {
        ...
    //@ spec_public
    private double amount = 0;
    //@ spec_public
    private Cabin cabin;
    //@ public invariant amount >= 0;
    //@ public invariant cabin == null ==> amount == 0;
}
```

(d) *(5 punten)* We korten af: $a = amount$, $c = cabin$. De te bewijzen eigenschap is gegeven door de Hoare-triple:

$$\{R \wedge I\}\ \texttt{resetCabin}\ \{E \wedge I\}$$

waarbij

$$\begin{aligned} R &= c \neq \mathsf{null} \wedge a = 0 \\ E &= c = \mathsf{null} \wedge c_0.cust = \mathsf{null} \\ I &= a \geq 0 \wedge (c = \mathsf{null} \Rightarrow a = 0) \end{aligned}$$

Het is gemakkelijk in te zien dat

$$\begin{aligned} I \wedge E &= c = \mathsf{null} \wedge c_0.cust = \mathsf{null} \wedge a = 0 \\ I \wedge R &= c \neq \mathsf{null} \wedge a = 0 \end{aligned}$$

$$\begin{aligned} & wp(\texttt{c0 = c; c.setCust(}\textbf{\texttt{null}}\texttt{); c=}\textbf{\texttt{null}}\texttt{;})\{c = \mathsf{null} \wedge c_0.cust = \mathsf{null} \wedge a = 0\} \\ = \ & wp(\texttt{c0 = c; c.setCust(}\textbf{\texttt{null}}\texttt{);})\{c_0.cust = \mathsf{null} \wedge a = 0\} \\ = \ & wp(\texttt{c0 = c;})\{c \neq \mathsf{null} \wedge (c_0 = c \vee c \neq c_0 \wedge c_0.cust = \mathsf{null}) \wedge a = 0\} \\ = \ & c \neq \mathsf{null} \wedge a = 0 \end{aligned}$$

Merk op dat in de op één na laatste stap het contract van `Cabin.setCust` gebruikt wordt. Bovendien wordt er een gevalsonderscheiding gemaakt gebaseerd op de aliasing van c and c_0: als $c = c_0$ dan vervalt de voorwaarde $c_0.cust = \mathsf{null}$ vanwege het contract van `Cabin.setCust`. *Deze vorm van aliasing is niet expliciet aan de orde geweest in het college!*

De laatste bewijsstap is triviaal, aangezien (zoals boven opgemerkt)

$$I \wedge R = c \neq \mathsf{null} \wedge a = 0$$

die triviaal waar is.