Tentamen Formele Methoden voor Software Engineering (192135201)

19 april 2012, 8:45–12:15 uur.

- Vermeld je studierichting op het tentamen
- Geef aan of je de huiswerkopgaven gemaakt hebt, en in welke groep/bij welke werkcollegeleider.
- Naast de sheets van de colleges mag je de FSP Quick Reference Card en de JML Cheat Sheet gebruiken.
- Het cijfer voor dit tentamen is gelijk aan het het behaalde aantal punten gedeeld door 10.

---

1. *(25 punten)* Beschouw de volgende FSP-specificatie:

```
// Ideale winkel
IAPC1 = (vraag -> koop -> IAPC1).

// Winkel waar niet alles op voorraad is
ZAAK2 = ( vraag -> ( koop -> ZAAK2 | bestel -> BESTEL) ),
  BESTEL = (wacht -> BESTEL | lever -> koop -> ZAAK2).

||IAPC2 = ZAAK2 \ {bestel, wacht, lever}.

// Winkel met parallel bestelproces
ZAAK3 = ( vraag -> ( koop -> ZAAK3 | bestel -> ZAAK3 )
        | lever -> koop -> ZAAK3
        ).
BESTEL3 = ( bestel -> lever -> BESTEL3 ).

||IAPC3 = (ZAAK3 || BESTEL3) \ {bestel, lever}.
```

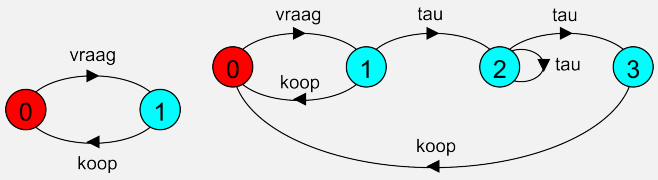
   (a) *(8 punten)* Teken de transitiesystemen van `IAPC1`, `IAPC2` en `IAPC3`.

   (b) *(7 punten)* Welke van de drie bovenstaande processen zijn bisimilair? Geef de relatie tussen toestanden aan bij bisimilaire processen, of leg anders uit waarom dit niet kan.

   (c) *(5 punten)* Voldoet `IAPC3` aan de standaard-*progress*-eigenschap? Let uit waarom, of waarom niet.

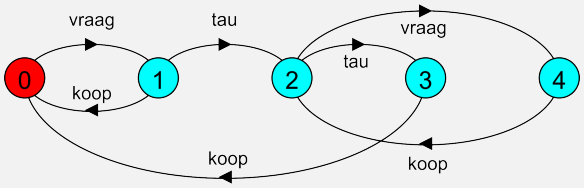
   (d) *(5 punten)* Formuleer een *safety property* die uitdrukt dat er uitsluitend afwisselend `vraag`- en `koop`-acties kunnen plaatsvinden, en teken het bijbehorende transitiesysteem. Welke van de bovenstaande processen voldoet niet aan deze eigenschap, en onder welke omstandigheden?

1. (a) (*8 punten*) De transitiesystemen:



IAPC1

IAPC2



IAPC3

(b) (*7 punten*) `IAPC1` en `IAPC2` zijn bisimilair: toestanden 1, 2 en 3 van `IAPC2` gedragen zich hetzelfde als toestand 1 van `IAPC1`, namelijk kan er alleen een `koop`-actie gebeuren.

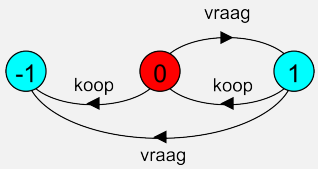
`IAPC3` is niet bisimilair met de andere processen: dit proces heeft namelijk een trace `vraag vraag` die de anderen niet hebben. De processen zijn dus niet eens trace equivalent.

(c) (*5 punten*) De standaard *progress*-eigenschap zegt dat onder de aanname van *fair choice* elke actie onbeperkt vaak zal gebeuren. Dit is hier het geval, want vanuit elke toestand blijven alle acties bereikbaar.

(d) (*5 punten*) De eigenschap lijkt erg op `IAPC1`, en luidt

```
property EERLIJK = (vraag -> koop -> EERLIJK).
```

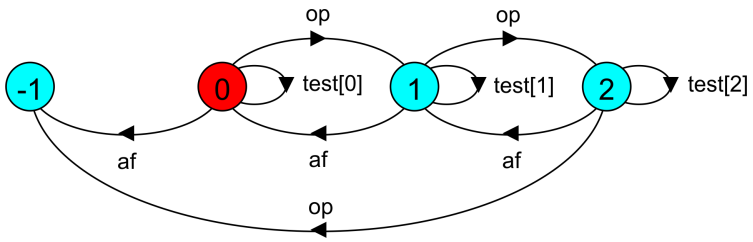
Het transitiesysteem is



Alleen `IAPC3` voldoet niet aan deze eigenschap, vanwege de bestaande trace `vraag vraag` die in `EERLIJK` naar de fouttoestand `-1` leidt.

2. *(25 punten)*

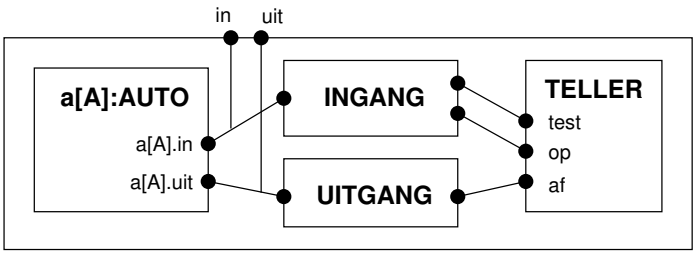
(a) (*8 punten*) Schrijf een process `TELLER` in FSP dat het volgende gedrag vertoont:



(b) (*10 punten*) Specificeer een parkeergarage met één ingang, één uitgang en een aantal auto's, waarbij:

- De ingang telkens test of de teller lager dan 2 staat, en zo ja, een auto toelaat en de teller ophoogt;
- De uitgang telkens een auto laat vertrekken en vervolgens de teller aflaagt;
- Een auto telkens de parkeergarage in- en uitrijdt.

Het systeem is schematisch in de volgende figuur weergegeven:



Geef FSP-definities voor `AUTO`, `INGANG`, en `UITGANG` en het samengestelde systeem.

(c) (*7 punten*) Is het samengestelde systeem veilig, in de zin dat er geen *error*-toestand bereikbaar is? Licht het antwoord toe.

2. (a) (*8 punten*) Het proces kan er bijvoorbeeld als volgt uitzien:

```
// Indexbereik van de teller
range I = 0..2
TELLER = TELLER[0],
  TELLER[i:I] = ( test[i] -> TELLER[i]
                | op -> TELLER[i+1]
                | af -> TELLER[i-1] ).
```

   (b) (*10 punten*) De processen en hun samenstelling:

```
// Auto-nummers
range A = 0..4

AUTO     = ( in -> uit -> AUTO ).
INGANG   = ( test[i:I] ->
             if (i<2) then (op -> a[A].in -> INGANG)
                      else INGANG ).
UITGANG  = ( a[A].uit -> af -> UITGANG ).

||SYS = (a[A]:AUTO || INGANG || UITGANG || TELLER) \ { test, op, af }.
```

   (c) (*7 punten*) Het gaat hier om de toestnd `-1` in `TELLER`. Bovenstaande oplossing geeft geen fouten, aangezien er maar één `INGANG` is en de acties `test` en `op` dus als atomair kunnen worden beschouwd. Als `op` en `a[A].in` worden omgedraaid gaat het wél mis, aangezien dan `op` kan worden uitgesteld tot de auto alweer de garage uitrijdt.

3. *(30 punten)* Beschouw de volgende Java-interface:

```
/** Periode in de geschiedenis.                                    *
  * Een tijdperk wordt bepaald door een begin- en een eindjaar *
  * met minstens een jaar verschil. Jaartallen zijn positief.  */
interface Tijdperk {
    /** Verandert het beginjaartal van dit tijdperk. */
    void setBegin(int begin);

    /** Verandert het eindjaartal van dit tijdperk. */
    void setEind(int eind);

    /** Levert het beginjaartal van dit tijdperk op. */
    int getBegin();

    /** Levert het aantal jaren van dit tijdperk op. */
    int getDuur();

    /** Levert de combinatie van dit tijdperk en een
      * direct aansluitend ander tijdperk op. */
    Tijdperk combinatie(Tijdperk ander);
}
```

(a) *(12 punten)* Stel een JML-contract op voor `Tijdperk`, waarin de beoogde werking zoveel mogelijk formeel gespecificeerd is. *Maak gebruik van modelvariabelen voor beginjaar en duur.*

```
interface Tijdperk {
    //@ instance model int begin;
    //@ instance model int duur;
    //@ instance invariant begin > 0 && duur > 0;

    //@ requires begin > 0;
    //@ ensures this.begin == begin;
    void setBegin(int begin);

    //@ requires eind > this.begin;
    //@ ensures this.duur == eind - this.begin;
    void setEind(int eind);

    //@ ensures \result == begin;
    int getBegin();

    //@ ensures \result == duur;
    //@ pure
    int getDuur();

    //@ requires this.eind == ander.begin;
    //@ ensures \result != null;
    //@ ensures \result.begin = this.begin;
    //@ ensures \result.duur = this.duur + ander.duur;
    //@ pure
    Tijdperk combinatie(Tijdperk ander);
}
```

Twee aaanvullende opmerkingen:

- Bovenstaand contract specificeert niet dat `duur` onveranderd blijftin `setBeing`, en `begin` in `setEind`. Dat zou eigenlijk wel moeten, maar in het college is nooit gerept over de noodzaak om expliciet te specificeren dat velden onveranderd blijven.

- Een andere denkbare oplossing is dat `setBegin` de duur van het interval verandert. Dat vergt een andere **`requires`** en een extra **`ensures`**.

Merk op dat het niet eenduidig vastligt

(b) *(8 punten)* Schrijf een implementatie voor `Tijdperk` *met velden voor het begin- en eindjaartal* en voorzien van alle additionele JML-specificaties die nodig zijn om te kunnen bewijzen dat de klasse correct is.

Een fout in onderdeel 3a kan natuurlijk hier doorwerken. Het belangrijkste zijn de JML-specificaties, bestaande uit de **`represents`**-clausules en de invarianten, inclusief de constructor.

```
class Implementatie implements Tijdperk {
    //@ requires begin > 0;
    //@ requires eind > begin;
    //@ ensures this.begin = begin;
    //@ ensures this.duur = eind-begin;
    Implementatie(int begin, int eind) {
        this.begin = begin;
        this.eind = eind;
    }

    public void setBegin(int begin) {
        this.eind = this.eind + (begin - this.begin);
        this.begin = begin;
    }

    public void setEind(int eind) {
        this.eind = eind;
    }

    public int getBegin() {
        return this.begin;
    }

    public int getDuur() {
        return this.eind - this.begin;
    }

    public Tijdperk combinatie(Tijdperk ander) {
        return new Implementatie(this.begin,
                                 ander.getBegin() + ander.getDuur());
    }

    private int _begin;
    private int _eind;
    //@ represents begin = _begin;
    //@ represents duur = _eind - _begin;
    //@ private invariant _eind > _begin;
}
```

(c) *(10 punten)* Bewijs de correctheid van de (zelf geïmplementeerde) methode `combinatie`.

After filling the representing expressions, and abbreviating `_begin` to b, `_eind` to e and `ander`

to a, we get

$$\begin{aligned} P &\equiv \mathsf{this}.e = a.b \\ Q &\equiv r \neq \mathsf{null} \wedge r.b = \mathsf{this}.b \wedge (r.e - r.b) = (\mathsf{this}.e - \mathsf{this}.b) + a.d \\ I &\equiv \mathsf{this}.b > 0 \wedge \mathsf{this}.e - \mathsf{this}.b > 0 \end{aligned}$$

(Note that a is not known to be of type `Tijdperk` and so we do not know anything about the representation of $a.d$.) The proof obligation is

$$\{P \wedge I\}\ \texttt{r = new I(this.b, a.getB() + a.getD());}\ \{Q \wedge I\}\ .$$

Since the method is pure we do not prove the postcondition I and just concentrate on proving Q.

$$\begin{aligned} &wp(\texttt{r = new I(this.b, a.getB() + a.getD());})\{Q\} \\ &\quad\equiv \mathsf{new}I(\mathsf{this}.b, a.getB() + a.getD()) \neq \mathsf{null} \\ &\qquad \wedge \mathsf{new}I(\mathsf{this}.b, a.getB() + a.getD()).b = \mathsf{this}.b \\ &\qquad \wedge \mathsf{new}I(\mathsf{this}.b, a.getB() + a.getD()).d = (\mathsf{this}.e - \mathsf{this}.b) + a.d \\ &\qquad \wedge \mathsf{this}.b > 0 \\ &\qquad \wedge a.getB() + a.getD() > \mathsf{this}.b \\ &\quad\equiv \mathsf{this}.b = \mathsf{this}.b \\ &\qquad \wedge (a.b + a.d) - \mathsf{this}.b = (\mathsf{this}.e - \mathsf{this}.b) + a.d \\ &\qquad \wedge \mathsf{this}.b > 0 \\ &\qquad \wedge a.b + a.d > \mathsf{this}.b \\ &\quad\equiv (\mathsf{this}.e = a.b) \wedge \mathsf{this}.b > 0 \wedge (a.b + a.d > \mathsf{this}.b) \end{aligned}$$

This is indeed implied by $P \wedge I$, in combination with the invariant of a:

$$\begin{aligned} P \wedge I &\equiv (\mathsf{this}.e = a.b) \wedge \mathsf{this}.b > 0 \wedge (\mathsf{this}.e > \mathsf{this}.b) \wedge a.d > 0 \\ &\equiv (\mathsf{this}.e = a.b) \wedge \mathsf{this}.b > 0 \wedge (a.b + a.d > \mathsf{this}.b) \end{aligned}$$

4. *(20 punten)* Het maximum van een `int[]`-array is een waarde x in de array zodat geen van de (andere) waarden in de array groter is dan x.

   (a) (*3 punten*) Formuleer in predicaatlogica en in JML de eigenschap dat `x` het maximum van `int[] a` is (*zonder* de speciale JML-syntax voor `\max` te gebruiken).

   (b) (*3 punten*) Geef een contract voor de volgende Java-methode dat uitdrukt dat de methode het maximum van zijn argument oplevert:

```
int max(int[] a) {
    int result = a[0];
    int i = 1;
    while (i < a.length) {
        if (a[i] > result) {
            result = a[i];
        }
        i++;
    }
    return result;
}
```

   (c) (*14 punten*) Bewijs dat de methode correct is.

4. (a) (*3 punten*) In predicate logic:

$$(\exists k : 0 \leq k \wedge k < |a| \wedge x = a[k]) \wedge (\forall k : 0 \leq k \wedge k < |a| \Rightarrow a[k] \leq x)$$

   In JML:

```
(\exists int k: k >= 0 && k < a.length; x == a[k])
&& (\forall int k: k >= 0 && k < a.length; a[k] <= x)
```

   (b) (*3 punten*) The method with contract and loop invariant:

```
/*@ requires a != null && a.length > 0;
  @ ensures (\exists int i: i >= 0 && i < a.length; \result == a[i])
  @      && (\forall int i: i >= 0 && i < a.length; a[i] <= \result);
  @*/
int max(int[] a) {
    int result = a[0];
    int i = 1;
    /*@ loop_invariant i <= a.length;
      @ loop_invariant (\exists int k: k>=0 && k<i: result == a[k]);
      @ loop_invariant (\forall int k: k>=0 && k<i: a[k] <= result);
      @*/
    while (i < a.length) {
        if (a[i] > result) {
            result = a[i];
        }
        i++;
    }
    return result;
}
}
```

   (c) (*14 punten. 10 Points can be earned with the correct invariant and a correct proof structure, including splitting the invariant proof to cope with the* `if`*-statement; the remaining 4 are awarded for proof details.*) We introduce two abbreviations, to express that a value occurs in an initial fragment of an array and to express that it is at least as large as all the elements in the array:

$$IsIn(a, n, x) \equiv \exists k : 0 \leq k \wedge k < n \wedge x = a[k]$$
$$Max(a, n, x) \equiv \forall k : 0 \leq k \wedge k < n \Rightarrow a[k] \leq x \ .$$

The five-step plan then gives rise to:

i. Converted pre-and postconditions:

$$P \equiv a \neq \mathsf{null} \wedge |a| > 0$$
$$Q \equiv \mathit{IsIn}(a, |a|, r) \wedge \mathit{Max}(a, |a|, r) \ .$$

ii. Loop invariant:

$$I \equiv i \leq |a| \wedge \mathit{IsIn}(a, i, r) \wedge \mathit{Max}(a, i, r) \ .$$

iii. Precondition implies invariant:

$$\begin{aligned} wp(\texttt{r=a[0]; i=1;})\{I\} &\equiv wp(\texttt{r=a[0];})\{1 \leq |a| \wedge \mathit{IsIn}(a, 1, r) \wedge \mathit{Max}(a, 1, r)\} \\ &\equiv wp(\texttt{r=a[0];})\{|a| > 0 \wedge r = a[0] \wedge r \geq a[0]\} \\ &\equiv |a| > 0 \end{aligned}$$

Clearly, $P \Rightarrow |a| > 0$, so we are done.

iv. Invariant is correct: first note that

$$\mathit{IsIn}(a, i+1, r) \equiv \mathit{IsIn}(a, i, r) \vee a[i] = r$$
$$\mathit{Max}(a, i+1, r) \equiv \mathit{Max}(a, i, r) \wedge r \geq a[i] \ .$$

$$\begin{aligned} &wp(\textbf{if}\ \texttt{(a[i]>r) \{r=a[i];\} i=i+1;})\{i \leq |a| \wedge \mathit{IsIn}(a, i, r) \wedge \mathit{Max}(a, i, r)\} \\ &\quad\equiv wp(\textbf{if}\ \texttt{(a[i]>r) \{r=a[i];\}})\{i < |a| \wedge \mathit{IsIn}(a, i+1, r) \wedge \mathit{Max}(a, i+1, r)\} \\ &\quad\equiv (a[i] > r \wedge wp(\texttt{r=a[i];}) \left\{ \begin{array}{l} i < |a| \\ \wedge\ (\mathit{IsIn}(a, i, r) \vee a[i] = r) \\ \wedge\ \mathit{Max}(a, i, r) \wedge r \geq a[i] \end{array} \right\} \\ &\qquad\vee (a[i] \leq r \wedge i < |a| \wedge (\mathit{IsIn}(a, i, r) \vee a[i] = r) \wedge \mathit{Max}(a, i, r) \wedge r \geq a[i]) \\ &\quad\equiv a[i] > r \wedge i < |a| \\ &\qquad\qquad \wedge (\mathit{IsIn}(a, i, a[i]) \vee a[i] = a[i]) \\ &\qquad\qquad \wedge \mathit{Max}(a, i, a[i]) \wedge a[i] \geq a[i] \\ &\qquad\vee a[i] \leq r \wedge i < |a| \wedge (\mathit{IsIn}(a, i, r) \vee a[i] = r) \wedge \mathit{Max}(a, i, r) \\ &\quad\equiv i < |a| \wedge (a[i] > r \wedge \mathit{Max}(a, i, a[i]) \\ &\qquad\qquad \vee a[i] \leq r \wedge (\mathit{IsIn}(a, i, r) \vee a[i] = r) \wedge \mathit{Max}(a, i, r)) \end{aligned}$$

Both branches of this condition are implied by $I \wedge i < |a|$.

v. Invariant plus negated loop condition establishes postcondition:

$$I \wedge i \geq |a| \quad \equiv \quad i = |a| \wedge \mathit{Max}(a, i, r) \quad \Rightarrow \quad \mathit{Max}(a, |a|, r) \ .$$