Tentamen Formele Methoden voor Software Engineering (192135201)

18 april 2013, 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. *(20 punten)* Beschouw de volgende FSP-specificatie:

```
// De gewenste gang van zaken
STUDIE1 =    ( geslaagd -> diploma -> STUDIE1 ).

// Opgedeeld in rollen
STUDENT2 = ( geslaagd -> controle -> STUDENT2 ).
BOZ =   ( controle -> diploma -> BOZ ).

||STUDIE2 = (STUDENT2 || BOZ) \ {controle}.

// Mogelijk (hoewel ongewenst) alternatief
DOCENT  = ( geslaagd -> DOCENT | gezakt -> DOCENT ).
STUDENT3 = ( geslaagd -> controle -> STUDENT3 | gezakt -> STUDENT3 ).

||STUDIE3 = (DOCENT || STUDENT3 || BOZ) \ {controle, gezakt}.
```

   (a) *(8 punten)* Teken de transitiesystemen van `STUDIE1`, `STUDIE2` en `STUDIE3`. Teken in de laatste twee gevallen ook de deelprocessen `STUDENT2`, `BOZ`, `STUDENT3`, en `DOCENT`.

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

2. *(30 punten)* De volgende eenvoudige FSP specificatie beschrijft een bibliotheek met een boek en een student.

```
BOOK    = ( taken -> returned -> BOOK).

STUDENT = ( taken -> study -> RETURN ),
    RETURN = ( returned -> STUDENT ).

||LIBRARY = (STUDENT || BOOK).
```

   (a) *(10 punten.)* Verander deze specificatie in een bibliotheek met drie studenten en drie boeken. Verder moet een student twee boeken bemachtigen voor hij kan gaan studeren. (*Hint:* Als een student boek i gepakt heeft ($i = 0, 1, 2$), moet-ie daarna nog boek $(i+1)\%3$ of $(i+2)\%3$ pakken.)

   (b) *(10 punten.)* Het uitgebreide proces `LIBRARY` (met drie studenten en drie boeken) bevat een deadlock. Geef een trace die naar een deadlock leidt. We kunnen dit verhelpen door studenten de mogelijkheid te geven boeken te laten terugleggen als er geen tweede boek beschikbaar is. Pas de specificatie zo aan dat dit correct gemodelleerd wordt.

   (c) *(10 punten.)* We willen ook graag dat elke student uiteindelijk twee boeken kan bemachtigen en kan gaan studeren. Druk deze eis uit m.b.v. een of meerdere **`progress`** eigenschappen.

   Maak aannemelijk dat deze eis niet wordt vervuld wanneer we twee dominante studenten hebben die, zodra er een boek vrijkomt, dat boek altijd eerder te pakken hebben dan de derde student. Hoe modelleren we dit scenario in FSP?

3. *(30 punten)* Beschouw de volgende Java-interface (waarbij `Team` een voorgedefinieerde klasse is).

```
interface Wedstrijd {
    /** Levert het thuisteam van de wedstrijd op. */
    Team getThuis();

    /** Levert het uitteam van de wedstrijd op. */
    Team getUit();

    /** Registreert dat het thuis- of uitteam een punt heeft gemaakt. */
    void punt(Team team);

    /** Levert de score van het thuis- of uitteam op. */
    int getScore(Team team);

    /** Levert de winnaar van de wedstrijd op, als die er is. */
    Team getWinnaar();
}
```

   (a) *(10 punten)* Stel een JML-contract op voor `Wedstrijd`, waarbij de beoogde werking zoveel mogelijk formeel gespecificeerd is. Maak waar dit nuttig is gebruik van modelvariabelen.

   (b) *(8 punten)* Schrijf een implementatie `NormaleWedstrijd` van bovenstaand interface, met voldoende JML-commentaar om de correctheid te kunnen bewijzen. Representeer daarbij de modelvariabelen door middel van het totaal aantal punten en het verschil tussen thuis- en uitteam, in plaats van door de aantallen punten per team.

   (c) *(7 punten)* Bewijs de correctheid van uw methode `punt`.

   (d) *(5 punten)* Geef een JML-invariant voor `NormaleWedstrijd` die uitdrukt dat het thuisteam nooit op verlies kan staan. Deze invariant geldt uiteraard niet. Leg precies uit, in termen van het JML-contract van de klasse, waar de fout zit.

4. *(20 punten)* De JML-expressie `(\sum int; P(i); E(i))` levert, voor een gegeven conditie `P` en expressie `E`, de som op van alle $E(i)$ waarvoor $P(i)$ geldt. Bijvoorbeeld is `(\sum int i; 1<=i && i<=n; i)` de som van alle gehele getallen van 1 tot en met n — oftewel, in wiskundige notatie is dit hetzelfde als $\sum_{i=1}^{n} i$.

   Beschouw nu de volgende methode, die het aantal voldoendes in een gegeven array `cijfers` oplevert:

```
int tel(int[] cijfers) {
    int result = 0;
    int k = 0;
    while (k < cijfers.length) {
        if (cijfers[k] > 5) {
            result = result + 1;
        }
        k = k+1;
    }
    return result;
}
```

   (a) (*5 punten*) Voorzie deze methode van een contract dat de beoogde werking zo precies mogelijk vastlegt. Hierbij is `cijfers` een array met cijfers, d.w.z., getallen van 1 tot en met 10.

   (b) (*10 punten*) Bewijs met behulp van een lusinvariant dat de methode aan haar contract voldoet.

   (c) (*5 punten*) Beschouw nu een variant `telAlle` op de methode `tel`, met signatuur `int[] telAlle(int[] cijfers)`, die voor eenzelfde parameter `cijfers` een nieuwe array oplevert (zeg `r`) zodat `r[k]` voor elke zinnige `k` het aantal cijfers hoger dan `k` bevat. (Bijvoorbeelt bevat `r[5]` het aantal voldoendes.) Geef een zo precies mogelijk contract voor deze variant. (N.B.: je hoeft de methode `telAlle` zelf niet uit te programmeren!)