EXAM
System Validation
(192140122)
13:45 - 16:45
02-11-2015

- This exam consists of 8 exercises.
- The exercises are worth a total of 100 points.
- The final grade for the course is $\frac{(hw_1+hw_2)/2+exam/10}{2}$, provided that you obtain at least 50 points for the exam (otherwise, the final grade for the course is at most 4).
- The exam is open book: all *paper* copies of slides, papers, notes etc. are allowed.

Exercise 1: Formal Tools and Techniques (10 points)

You work for a company that develops an app called MediCare. The app has several relatively separate components:

- a fun-to-play movement game, to make patients have sufficient daily movement;
- an agenda that warns patients when it is time to take medications, and asks for a confirmation that the medication has been taken; if this confirmation is not given it sends a message to the doctor of the patient; and
- a heart and blood pressure measuring tool.

The company has limited budget, both in time and money, to ensure reliability and quality of their app, but they realise that reliability is very important for their credibility. As you are the expert in software reliability in the company, you are asked to give an advice on how to spend the budget on improving software reliability:

- how should the time and money be spend?
- where should the major efforts go?
- what sort of techniques should be used?
- what goals should be aimed for?

Give your advice in approximately 200 words.

Answer

Of course, there is not a single possible solution here. Important ingredients are in my opinion:

- for game reliability much less important than for medication agenda and signalling function to doctor
- if the system forgets to send a signal to the doctor, a patient might die of that, so the second component can be considered safety-critical.

- major efforts should go to the second component: build a model to check high-level system behaviour, use static checking (and maybe software model checking, if properties appropriate) to check actual implementation
- for game, non-crashing is probably sufficient property
- for measuring tool, reasonably accuracy of the results is necessary
- for medication tool, ensure properties such as: notification is never missed, every time there is no confirmation, a message is send to the doctor.

Exercise 2: Specification (10 points)

Write formal specifications for the following informal requirements in an appropriate specification formalism of your choice. You may assume that appropriate atomic propositions, query methods and classes exist.

a (*3 points*) If you order a game online, eventually it will be delivered.

b (*4 points*) If you order a book online, there always is a possibility that the package will be lost in the mail.

c (*3 points*) There are always at least 10,000 books that have been ordered, but have not been delivered yet.

Answers

Many different formalisations can exist. Also in several cases, both temporal logic and a JML specification would be an appropriate choice. (Thus answers that differ from the answers below might still be correct.)

a $\mathsf{G}\,(\mathrm{order} \rightarrow \mathsf{F}\,\mathrm{delivered})$

b $\mathsf{A}\,\mathsf{G}\,(\mathrm{order} \rightarrow \mathsf{E}\,\mathsf{F}\,\mathrm{lost})$

c invariant ordered − delivered >= 10000;

Exercise 3: Modelling (15 points)

Consider an automated bike parking such as in Enschede station. There are two doors: an entry for persons, and an entry for bikes. At both sides of the doors, there is card reader. When users of the bike parking arrive, they enter their card in the card reader. If the card processing succeeds, one or two doors open, depending on whether there is only a person, or also a bike. If there are users on both sides of the doors, only one of them will be allowed to enter/leave, the other user will have to wait.

A model for a person interacting with a card reader at the door is given:

```
1 MODULE Reader(
2   person_door, -- boolean true iff the person door is open
3   cycle_door, -- boolean true iff the cycle door is open
4   pass -- boolean true iff passing is allowed
5 )
6 VAR
7   state : { idle, allow };
8   cycle : boolean; -- person has cycle with him/her
9
10 ASSIGN
11   init(state):= idle;
12   next(state):= case
13     -- when idle choose what to do next.
14     state = idle : { idle, allow };
15     -- when all necessary doors are open and we have permission
16     -- go through the door and thus complete the action
```

```
17      state=allow & person_door & (cycle -> cycle_door) & pass : idle;
18      -- otherwise wait and do nothing
19      TRUE:state;
20    esac;
21
22    -- choose whether or not to take a bicycle
23    next(cycle):=case
24      state!=allow & next(state)=allow : {TRUE,FALSE};
25      TRUE:cycle;
26    esac;
```

a (*6 points*) Write an SMV model of the control system for the two doors.

It should declare two variables of type Reader:

```
1 MODULE main
2
3 VAR
4   inside : Reader(...);
5   outside : Reader(...);
6
7 ...
```

Otherwise you are free to declare your own variables.

The behaviour of the system should account for the following:

- The bicycle door should only open if the person door is opened for someone who has a bicycle with him/her. (Note the cycle variable in Reader).
- When people are waiting on both sides, the person to go through the door shall be selected non-deterministically and the appropriate doors shall be opened.
- The system opens a door if a reader allows it and closes the door when the reader no longer allows it.

You may use abbreviations, *e.g.*, if a long formula occurs several times you may mark the first occurrence with a name and then use this name for subsequent occurrences of the formula.

b (*3 points*) Specify that the bike door never can be opened without the people door also being opened.

c (*3 points*) Specify that it is always possible to leave the bike parking.

d (*3 points*) We wish to ensure that the model contains no runs in which a user is waiting to leave the bike parking, but that all the time users that are entering are being served, and thus the user has to stay in the bike parking for ever. What needs to be changed in the model to ensure this?

Answer

```
MODULE main

DEFINE
  C := 10;

VAR
  inside : Reader(person_door, cycle_door, pass_inside);
  outside : Reader(person_door, cycle_door, pass_outside);
  pass_inside : boolean;
  pass_outside : boolean;
  person_door : boolean;
  cycle_door : boolean;
  select : {idle, inside, outside, closing};

ASSIGN
  init(pass_inside) := FALSE;
  init(pass_outside) := FALSE;
  init(person_door) := FALSE;
  init(cycle_door) := FALSE;
  init(select) := idle;
  next(select) := case
    select=inside & inside.state != allow : closing;
    select=outside & outside.state != allow : closing;
    select=closing & !person_door & !cycle_door : idle;
    select=idle & inside.state=allow & outside.state=allow : {inside, outside};
    select=idle & inside.state=allow : inside;
    select=idle & outside.state=allow : outside;
    TRUE: select;
  esac;
  next(pass_inside):=select=inside;
  next(pass_outside):=select=outside;
  next(person_door) := select in {inside, outside};
  next(cycle_door) := case
    select=outside & outside.cycle : TRUE;
    select=inside  & inside.cycle : TRUE;
    TRUE : FALSE;
  esac;

-- (b)
LTLSPEC G (cycle_door -> person_door)

-- (c)
SPEC AG (inside.state=allow -> EF inside.state=idle)

-- (d)
FAIRNESS inside.state!=allow
```

Exercise 4: Software Model Checking (15 points)

Consider the following small code fragment:

```
1 if (x == 1) {
2   x = x + 1;
3 } else {
4   x = 2;
5 }
```

a (*3 pnts*) How to specify that whatever the program does, it will end in a state where x == 2?

b (*6 pnts*) If the program is annotated with an assertion that x == 2 then predicate abstraction will start with this predicate. What is the resulting Boolean program (including the abstract assertion)?

c (*6 pnts*) Explain what happens when the Boolean program is checked, and how we can conclude that x == 2 indeed always holds.

Answers

a (*3 pnts.*) An assertion assert x == 2 after the complete if-then-else statement.

b (*6 pnts.*)

```
1 p = *;
2 if ( p ? F : * ) {
3    p = p ? F : *;
4 } else {
5   p = T;
6 }
7 assert p;
```

c (*6 pnts.*) Spurious counter example, and the abstraction will be refined to the two predicates x == 2 and x == 1, after which the it can be deduced that the property will hold.

Exercise 5: Abstraction (10 points)

Consider the interface Sensor and its implementations EUSensor and USSensor.

Add specifications to this interface and to the classes, in such a way that we can show that the classes correctly implement the interface. That is, that the temperature measured by the sensor, in degrees Celsius, never decreases below absolute zero (-273°C stored in the constant ZERO).

```
1 public interface Sensor {
2
3   public final int ZERO = -273;
4
5   public int getTemperature();
6
7 }
```

```
1 public class EUSensor implements Sensor {
2
3   private int tempInCelsius;
4
5   public int getTemperature(){
6     return tempInCelsius;
7   }
8 }
```

```
1 public class USSensor implements Sensor {
2
3   private int tempInFahrenheit;
4
5   private int convertToCelsius() {
6     return ((tempInFahrenheit - 32)*10)/18;
7   }
8
9   public int getTemperature() {
10    return convertToCelsius();
11  }
12 }
```

Answer

```
1 package sensor;
2
3 public interface Sensor {
4
5   public final int ZERO = -273;
6
7   //@ public instance invariant ZERO <= temp;
8   //@ public instance model int temp;
9
10  /*@
11     ensures \result==temp;
12     pure
13   @*/
14  public int getTemperature();
15
16 }
```

```
1 package sensor;
2
3 public class EUSensor extends Sensor {
4
5   //@ public represents temp  = tempInCelcius;
6
7   private /*@ spec_public @*/ int tempInCelcius;
8
9   public int getTemperature() {
10    return tempInCelcius;
11  }
12 }
```

```
1 package sensor;
2
3 public class USSensor extends Sensor {
4
5   //@ represents temp <- getTemperature();
6
7   public int getTemperature() {
8     return convertToCelcius();
9   }
10
11  private int tempInFahrenheit;
12
13  // @ spec_public
14  private /*@ pure */ int convertToCelcius() {
15    return ((tempInFahrenheit - 32)*10/18);
16  }
17
18 }
```

Exercise 6: Run-time Checking (15 points)

In this exercise, we consider a few classes that model traveling by air.

```
1  class Destination {
2    public final boolean usa;
3    public final String city;
4    //@ ensures this.usa==usa && this.city==city;
5    public Destination(String city,boolean usa){
6      this.usa=usa;
7      this.city=city;
8    }
9  }
10
11 class Trip {
12
13   public final Destination dest;
14   public final int price;
15   /*@ spec_public @*/ private int payed;
16
17   //@ public invariant price > 0;
18
19   //@ requires price > 0;
20   //@ ensures this.dest==dest && this.price==price && this.payed==0;
21   public Trip(Destination dest,int price){
22     this.dest=dest;
23     this.price=price;
24   }
25
26   public void pay(int amount){
27     payed=payed+amount;
28   }
29
30   public boolean applyESTA(){
31     // just assume it is done correctly.
32   }
33
34   public void travel(){
35     // just assume it is done correctly.
36   }
37 }
38
39
40 class Traveler {
41
42   public final Trip trip;
43
44   //@ public invariant trip.payed <= trip.price;
45
46   //@ requires trip.payed==0;
```

```
47   //@ ensures this.trip==trip;
48   public Traveler(Trip trip){
49     this.trip=trip;
50   }
51
52   /*@
53    requires trip.payed==0;
54    ensures  true;
55    */
56   public void take_trip(){
57     // confirm by paying 100.
58     trip.pay(100);
59     do_something();
60     trip.pay(trip.price-100);
61     do_something();
62     trip.travel();
63   }
64
65   //@ modifies \nothing;
66   public void do_something(){
67     // irrelevant what is done.
68   }
69
70   public static Destination NewYork=new Destination("New York",true);
71   public static Destination Barcelona=new Destination("Barcelona",false);
72
73   public static void test1(){
74     Traveler t=new Traveler(new Trip(NewYork,499));
75     t.take_trip();
76   }
77   public static void test2(){
78     Traveler t=new Traveler(new Trip(Barcelona,99));
79     t.take_trip();
80   }
81
82 }
```

Some specifications have already been provided, but there are additional requirements:

- Before taking any trip, it must be payed in full.
- When traveling to the USA, one must successfully complete an ESTA application.

a (*5 points*) Add (ghost) specifications to the `Trip` class that ensure the above properties.

b (2×5 *points*) Explain what exactly happens when the two test methods `test1` and `test2` are executed with the Runtime Assertion Checker and the specifications you have provided.

Answers

a (*5 pnts.*) The important parts of the specification are:

```
1   /*@
2     ghost public boolean esta_complete=false;
3
4     ensures \result ==> esta_complete;
5    */
6   public boolean applyESTA(){
7     // just assume it is done correctly.
8     //@ set esta_complete=true;
9     return true;
10  }
11
12  //@  requires payed >= price;
13  //@  requires dest.usa ==> esta_complete;
14  public void travel(){
15    // just assume it is done correctly.
16  }
```

b (*2 * 5 pnts.*) The results of RAC are:

test1 Failure when the trip is taken because the ESTA application has not been carried out.

test2 Failure because paying 100 violates the invariant of Traveler that says that he never overpays.

Exercise 7: Static Checking (15 points)

Consider class Skater, which models a skater making his laps. Everytime he crosses the finish line, the current time will be added to the array with lap times.

a *(5 points)* Add a loop invariant that should be sufficient to prove correctness of the method skateDistance.

b *(2∗5 points)* Find two problems that the Extended Static Checker will stumble on in this example.

In case you do not remember, the Java method System.currentTimeMillis() returns the current time in milliseconds, where 0 stands for midnight, January 1, 1970 UTC. Also long works just like int but it avoids overflows.

```
1  package skate;
2
3  public class Skater {
4  //@ public invariant rounds >= 0;
5  //@ public constraint \old(rounds) <= rounds;
6
7    private /*@ spec_public */ int rounds;
8    private /*@ spec_public */ long[] laptimes;
9
10   //@ ensures rounds == 0;
11   public void start() {
12     rounds = 0;
13     laptimes[0] = System.currentTimeMillis();
14   }
15
16   //@ modifies \everything;
17   //@ ensures rounds == \old(rounds) + 1;
18   //@ ensures \result > \old(laptimes[rounds]);
19   public long completeRound() {
20     long res = System.currentTimeMillis();
21     //@ assume res > laptimes[rounds];
22     rounds++;
23     return res;
24   }
25
26   //@ requires distance % 400 == 0;
27   /*@ ensures (\forall int i, j;
28         0 <= i && i < j && j < laptimes.length;
29         laptimes[i] < laptimes[j]); */
30   public void skateDistance(int distance) {
31     int laps = distance/400;
32     laptimes = new long[laps];
33     start();
34     while (rounds < laps) {
35       long t = completeRound();
36       laptimes[rounds] = t;
```

```
37          }
38      }
39  }
```

Answers

a The loop should be annotated in the following way:

```
1 /*@ loop_invariant
2           0 <= rounds && rounds <= laps &&
3           laps + 1 == laptimes.length &&
4           (\forall int i, j; 0 <= i && i < j && j <= rounds;
5                 laptimes[i] < laptimes[j]);
6 */
```

b Ther are many problems that ESC would find...

- In start: the first assignment breaks the **constraint on rounds only increasing**.
- In start: the second assignment could assign a non-existing element of the array if that array has length 0.
- In skateDistance: the creation of a new array might fail if laps is negative, so distance must be greater than 0.
- The assignable clauses of both `start` and `completeRounds` are too general and will not allow ESC to assume that `laptimes` is not changed which is necessary to prove the `skateDistance` method correct.
- In skateDistance: the assignment to laptimes[round] can fail because the array is one too element too short.

Exercise 8: Test Generation with JML (10 points)

Below you find a simple implementation of a semaphore The final field size contains the number of tokens that the semaphore controls. We consider the release method of the class. It can release several tokens at once.

a (*2 points*) The class needs one invariant to be maintained, specify it.

b (*5 points*) Provide **one** complete specification case for the normal execution of the release method, that is, when everything goes fine and the number of available tokens is increased by count without any exceptions being thrown. Specify preconditions and changes to the state suitable for generating a meaningful test case for this usage scenario.

c (*3 points*) For the complete testing, including all the exceptional cases, of the method release there would be more specification cases needed. How many exactly for this method? Provide at least one set of test data for each of these specification cases.

```
1 public final class Semaphore {
2
3   /** The (positive) number of semaphore tokens.
4    */
5   public final int size;
6
7   /** The number of tokens available for acquiring.
8    */
9   private int available;
10
11  /**
12   * Release count sempahore tokens at once.
13   */
14  public synchronized void release(int count)
15    throws InconsistentStateException
16  {
17    if(count <= 0) throw new IllegalArgumentException();
18    if(this.available + count > size) throw new InconsistentStateException();
19    this.available += count;
20  }
21
22  public Semaphore(int size){
23    this.size=size;
24  }
25 }
```

Answers

a The invariant is:

```
1 //@ invariant 0 <= available && available <= size;
```

```
1 normal_behaviour
2 requires 0 < count && count <= size - available;
3 ensures  available == \old(available) + count;
```

b There shall be 3 specification cases all together due to the three different execution branches that the code can take (two exceptional and one normal execution branch). The corresponding example test data would be:

	this.available	this.size	count
IllegalArgumentException	4	10	0
InconsistentStateException	8	10	4
normal case	4	10	4