EXAM
System Validation
(192140122)
13:45 - 16:45
21-01-2016

- 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, and notes are allowed. Copies of old exams are *not* allowed.

Exercise 1: Formal Tools and Techniques (10 points)

Suppose that you work at NASA as part of the MarsLander project. You are asked for advice on the development of software for a new project to bring people to Mars. The flight is expected to take about 18 months. Of course, the crew members should return to Earth safely.

You have to advise about the following components of the space craft control system:

- the aircraft control system, both for flying to Mars, where small route corrections are possible during the flight;
- the landing and take-off control system, for landing on Mars, and taking off later, to fly back to Earth;
- the climate control system, including the oxygen control; and
- the home entertainment system, that the crew can use to relax a bit when they are off duty.

Give your advise, using *no more than 200 - 250 words*, describing what are the desired properties for the two components and how can the software developers ensure that the components indeed satisfy these properties.

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*) A train may only cross a road when the barrier is closed.

(b) (*3 points*) The remaining mortgage debt should always decrease, until the mortgage is paid off.

(c) (*4 points*) After a change of plan, in the future it is always possible to change the plan again.

Exercise 3: Modeling (15 points)

Suppose we have N *flipping* processes modelled as a ring. Each flipping process receives its left and right neighbour as a parameter, and it has a colour: black or white. The colour changes as follows: if both neighbours have an equal colour, the colour becomes black, otherwise it is picked at random.

(a) (6 points) Describe the flipping process as a NuSMV module.

(b) (3 points) Describe the main process, initialising **5** flipping processes. The main process should be able to determine whether all flipping processes have the same colour.

(c) (3 points) How can it be ensured that a flipping process will not always be coloured black.

(d) (3 points) Specify the following property: if all colours are the same, they will remain the same forever.

Exercise 4: Software Model Checking (15 points)

Consider the following small code fragment:

```
1  unsigned int nondet_unsigned_int();
2
3  int main(){
4    unsigned int x = nondet_unsigned_int();
5
6    if (x > 1000) {
7      x = 481;
8    }
9
10   x = x + 37;
11
12   while( x > 4 ) {
13     x = x - 1;
14   }
15
16 }
```

(a) (*3 pnts*) How to specify that whatever the program does, it will end in a state where x == 4?

(b) (*6 pnts*) Assuming that predicate abstraction starts with a single predicate x > 4, what is the resulting Boolean program (including the abstract assertion)?

(c) (*6 pnts*) Why is the given abstraction not good enough to prove that the assertion in the main program never fails?
Which predicate(s) should be added to make the verification work?

Exercise 5: Abstraction (10 points)

Consider the interface Tank and its implementations BasicTank and DoubleTank.

Add specifications to this interface and to the classes, in such a way that we can show that the classes correctly implement the interface. (You can refer to line numbers to indicate where your specifications must be added.)

```
1  interface Tank {
2  /**
3   * Fill tank with up to amount fuel.
4   * This method returns the amount of fuel added.
5   * If the added amount is less that the given amount then
6   * the tank is full.
7   */
8  int fill(int amount);
9
10 /**
11  * Consume up to the given amount of fuel.
12  * This method return the amount of fuel that the tank was able to provide
13  * if this amount is less than the requested amount then the tank is empty.
14  */
15 int consume(int amount);
16 }
17
18 public class BasicTank implements Tank {
19   private int capacity;
20   private int stored=0;
21
22   public BasicTank(int capacity){
23     if (capacity<=0) throw new IllegalArgumentExecption();
24     this.capacity=capacity;
25   }
26
27   int fill(int amount){
28     if (amount<0) throw new IllegalArgumentExecption();
29     if (stored+amount >= capacity){
30       amount=capacity-stored;
31     }
32     stored=stored+amount;
33     return amount;
34   }
35
36   int consume(int amount){
37     if (amount<0) throw new IllegalArgumentExecption();
38     if (stored < amount){
39       amount=stored;
40     }
41     stored=stored-amount;
42     return amount;
43   }
```

```
44 }
45
46 public class DoubleTank implements Tank {
47
48   private Tank tank1;
49   private Tank tank2;
50
51   public DoubleTank(Tank t1,Tank t2){
52     if (t1==null || t2==null) throw new IllegalArgumentExecption();
53     tank1=t1;
54     tank2=t2;
55   }
56
57   int fill(int amount){
58     int tmp=tank1.fill(amount);
59     if (tmp<amount){
60        tmp=tmp+tank2.fill(amount-tmp);
61     }
62     return tmp;
63   }
64
65   int consume(int amount){
66     int tmp=tank1.consume(amount);
67     if (tmp<amount){
68        tmp=tmp+tank2.consume(amount-tmp);
69     }
70     return tmp;
71   }
72 }
```

Exercise 6: Run-time Checking (15 points)

In this exercise, we consider a few classes that model the order management system of an online shop.

```
1  public class Item {
2
3    public final String name;
4
5    public final int quantity;
6
7    public final int price;
8
9    public int reserved=0;
10
11   public Item(String name,int quantity,int price){
12     this.name=name;
13     this.quantity=quantity;
14     this.price=price;
15   }
16
17 }
18
19 public class Order {
20
21   private /*@ spec_public @*/ Item items[];
22
23   /*@
24     public invariant (\forall int i ; 0 <= i && i < items.length ;
25       (\forall int j ; 0 <= j && j < items.length ;
26         i != j ==> !items[i].name.equals(items[j].name)
27       )
28     );
29    */
30
31   public Order(Item ... list){
32     items=list;
33   }
34
35   /*@
36      ensures \result==(\sum int i; 0 <= i && i < items.length;items[i].quantity
37    */
38   /*@ pure @*/ public int total(){
39     int total=0;
40     for(int i=0;i<items.length;i++){
41       total=total+items[i].price;
42     }
43     return total;
44   }
45
```

```
46
47    public boolean complete(){
48      boolean complete=true;
49      int i=0;
50      while(complete && i < items.length){
51        if (items[i].reserved!=items[i].quantity){
52          complete=false;
53        }
54        i=i+1;
55      }
56      return complete;
57    }
58
59    public static void test1(){
60      Order order=new Order(
61        new Item("Pen",1,2),
62        new Item("Notebook",1,3)
63      );
64      System.out.printf("cost is %d%n",order.total());
65    }
66
67    public static void test2(){
68      Order order=new Order(
69          new Item("Pen",1,2),
70          new Item("Pen",1,2),
71          new Item("Notebook",1,3)
72      );
73      System.out.printf("cost is %d%n",order.total());
74    }
75  }
```

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

- Price and quantity are positive.
- The number of reserved items grows until it matches the quantity ordered.

(a) (*5 points*) Add (ghost) specifications to the `Item` 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.

Exercise 7: Static Checking (15 points)

(a) *(5 points)* Consider the method complete in the class Order that determines if an order is complete (the correct amount of all items in the order has been reserved).
Add a method contract and loop invariant that express this and that can be verified using the static checker.

(b) *(5 points)* Which problem(s), if any, will the static checker find if it is run on the following class Score:

```
1  public class Score {
2    /*@ spec_public */ private int participants;
3    /*@ spec_public */ private int[] score;
4
5    //@ public invariant score != null;
6    //@ public invariant participants == score.length;
7    /*@ public invariant (\forall int i; 0 <= i && i < participants; score[i]
8     */
9
10   //@ assignable \everything;
11   //@ requires nr >= 0;
12   public Score(int nr) {
13     participants = nr;
14     score = new int[nr];
15     clearScore();
16   }
17
18   /*@ ensures (\forall int i; 0 <= i && i < participants; score[i] == 1);
19    */
20   public void clearScore() {
21     int i=0;
22     while(i < participants)
23     {
24       score[i] = 1;
25       i = i + 1;
26     }
27   }
28 }
```

(c) *(5 points)* Which problem(s), if any, will the static checker find if it is run on the following class Factory:

```
1  public class Factory {
2    /*@ spec_public */ private int size=0;
3    /*@ spec_public */ private int employees=0;
4    /*@ spec_public */ private int production=0;
5
6    /*@ assignable production;
7        ensures production > \old(production);
8     */
9    public void produce() {
10     production=production+1;
11     if (production > 1000) {
12       expand ();
13     }
14   }
15
16   /*@ assignable size, employees;
17    */
18   public void expand() {
19     size = 2 * size;
20     employees = 2 * employees;
21   }
22 }
```

Exercise 8: Test Generation with JML (10 points)

Below you find part of the implementation of an in-memory file system. This file system is backed by a hash map. (Remember that in Java both the key and the data element of every entry in a hash map have to be non-null.)

(a) (*5 points*) Provide **one** complete specification case for the normal execution of the read 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.

(b) (*5 points*) For the complete testing, including all the exceptional cases, of the method read 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. (A human readable notation for hash map values suffices.)

```
1  import java.util.HashMap;
2  import java.util.Map;
3
4  public class MapFileSystem {
5
6    private /*@ spec_public @*/ Map<String,byte[]> map;
7
8    /*@
9     ensures \result==(map.containsKey(file)?map.get(file).length:-1);
10   @*/
11   public /*@ pure @*/ int size(String file){
12     return map.containsKey(file)?map.get(file).length:-1;
13   }
14
15   public int read(String file,byte[] data)
16     throws FileNotFoundException, ArrayTooSmallException
17   {
18     if(!map.containsKey(file)){
19       throw new FileNotFoundException();
20     }
21     byte[] contents=map.get(file);
22     if(contents.length>data.length){
23       throw new ArrayTooSmallException();
24     }
25     System.arraycopy(contents, 0, data, 0, contents.length);
26     return contents.length;
27   }
28 }
```