EXAM
System Validation
(192140122)
13:45 - 16:45
07-11-2016

- This exam consists of 7 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 (15 points)

Suppose that you work for a large software company. The company wants to increase their industrial impact by increasing the reliability of their software. They are interested in the use of formal methods and at a scientific workshop they learned about JML. They ask *you* for advice on how they could use JML to increase their industrial impact. Give your advice, using *no more than 200 - 250 words*. Your advice should at least address the following:

- What tools and techniques are readily available?
- How to introduce JML (and its capabilities) to their developers?
- What techniques should the company use for different language aspects?
- What additional tool support would they have to develop themselves?
- What aspects of JML would be the most beneficial for them?

Exercise 2: Specification (15 points)

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

(a) (*3 points*) Books are available or borrowed, but never both at the same time.

(b) (*3 points*) If a book is borrowed by person X, it first has to be returned to the library before it can be borrowed by person Y.

(c) (*3 points*) *Book life cycle:* A new book may enter the collection, may be borrowed a 1000 times, and then it is written off.

(d) (*3 points*) If a book is borrowed by someone, there always is the possibility it will not be returned to the library.

(e) (*3 points*) You may only borrow a book if you have paid your subscription fee.

Exercise 3: Abstraction (10 points)

Consider the Time class given below. Write an *abstract* specification of this class, defining the behaviour of the class in terms of a model variable timeInSeconds. Make the connection with the concrete variables explicit. You can refer to line numbers to indicate where your specifications must be added.

```
1  public enum Unit {
2          Seconds,
3          Minutes,
4          Hours
5  }
6
7  public class Time {
8          private Unit unit;
9          private int count;
10
11         //@ public model int timeInSeconds;
12
13         public Time(Unit unit) {
14                 this.unit = unit;
15                 this.count = 0;
16         }
17
18         public double timePassedInSeconds() {
19                 if (unit == Unit.Seconds) {
20                         return count;
21                 } else if (unit == Unit.Minutes) {
22                         return 60*count;
23                 } else {
24                         return 3600*count;
25                 }
26         }
27
28         public void addHours(int n) {
29                 if (unit == Unit.Seconds) {
30                         count = count + 3600*n;
31                 } else if (unit == Unit.Minutes) {
32                         count = count + 60*n;
33                 } else {
34                         count = count + n;
35                 }
36         }
37 }
```

Exercise 4: Software Model Checking (10 points)

In this exercise, we consider a program that models a lock. Specifically, the two vertical doors of the lock are modelled with an integer each that stores the height of the lock above the bottom. Height 0 means closed and height 8 means fully open:

```
1 #define closed 0
2 #define open 8
3
4 unsigned int door1=0;
5 unsigned int door2=0;
6 unsigned int counter=0;
7
8 #include "update.c"
9
10 int main(int argc,char*argv[]){
11   for (;;){
12     update();
13     assert(door1==closed || door2==closed);
14     sleep(1);
15     counter=counter+1;
16   }
17 }
```

This program has already been annotated to show that at least one of the doors of a lock is always closed after an update.

(a) (*3 points*) Without knowing how the file update.c implements the update procedure, annotate the main body to verify that the first door (represented by door1) runs in a sequence:

$$\text{closed} \rightarrow \text{going up} \rightarrow \text{open} \rightarrow \text{going down} \rightarrow \cdots$$

Note that update is not required to change the state of each door on every call.
Hint: use auxiliary variables to keep track of

- the state of the door before the call to update and
- the expected direction.

See next page.

(b) (*4 points*) We will try to prove the given claim using the following two predicates:

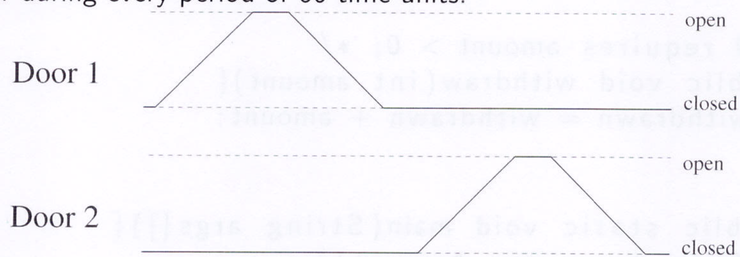
P1 door1==closed

P2 door2==closed

Below is the abstraction of the main program over these two predicates:

```
boolean P1,P2=T,T;
for (;;){
    update();
    assert(P1||P2); // assert(door1==closed || door2==closed);
    P1,P2 := P1,P2; // sleep(1);
    P1,P2 := P1,P2; // counter=counter+1;
}
```

As a strategy for opening, that is, one concrete implementation of update, we consider the following behaviour during every period of 60 time units:



Derive the boolean abstraction of the update function, given that it has the following implementation:

```
1  #define period 60
2
3  void update(){
4    if (0 < (count % period) &&
5          (count % period) < 9) {
6      // open during 0..9 units of the first half of the period
7      door1++;
8    } else if ((period/2) < (count % period) &&
9                            (count % period) < (period/2) + 9) {
10     // open during 0..9 units of the second half of the period
11     door2++;
12   } else if (15 < (count % (period/2)) &&
13                   (count % (period/2)) < 25){
14     // keep closing both during units 15..25 of every half-period.
15     door1=(door1==closed)?door1:(door1-1); // decrease if not closed.
16     door2=(door2==closed)?door2:(door2-1); // decrease if not closed.
17   }
18 }
```

(c) (*3 points*) Why is the given abstraction not good enough to prove that the assertion in the main program never fails?

Exercise 5: Run-time Checking (15 points)

Explain your answers (without explanation no points will be awarded).

(a) (*5 points*) What will happen when the JML run-time checker is used to validate the following class?

```
1  public class Account {
2    /*@ spec_public */ private int deposited=0;
3    /*@ spec_public */ private int withdrawn=0;
4
5    /*@ invariant deposited >= withdrawn ; */
6
7    /*@ requires amount > 0; */
8    public void deposit(int amount){
9      deposited = deposited + amount;
10   }
11
12   /*@ requires amount > 0; */
13   public void withdraw(int amount){
14     withdrawn = withdrawn + amount;
15   }
16
17   public static void main(String args[]){
18     Account a = new Account();
19     a.deposit(10);
20     a.withdraw(10);
21     a.withdraw(4);
22   }
23 }
```

(b) (*5 points*) What will happen when the JML run-time checker is used to validate the following class?

```
1  public class Pair {
2    public final int x, y;
3
4    /*@ ensures this.x==x && this.y==y; */
5    public Pair(int x, int y){
6      this.x=x;
7      this.y=y;
8    }
9
10   /*@ ensures \result <==> x > 0 && y > 0; */
11   public /*@ pure @*/ boolean firstQuadrant(){
12     return (x >= 0 && y >= 0);
13   }
14
15   public static void main(String args[]){
16     Pair p=new Pair(1,1);
17     boolean tmp=p.firstQuadrant();
```

```
18   }
19 }
```

(c) (*5 points*) What will happen when the JML run-time checker is used to validate the following class?

```
1  public class Buffer {
2    /*@ spec_public @*/ private int used=0;
3    /*@ spec_public @*/ private int size=16;
4    /*@ spec_public @*/ private byte[] data=new byte[16];
5
6    /*@ invariant data != null && data.length==size &&
7                 0 <= used && used < size; @*/
8
9    /*@ normal_behavior
10     @ requires size > 0;
11     @ ensures data.length == size; */
12   public Buffer(int size){
13     data=new byte[size];
14   }
15
16   /*@ normal_behavior
17     @ ensures \result <==> used==size; */
18   public /*@ pure @*/ boolean full(){
19     return used==size;
20   }
21
22   /*@ normal_behavior
23     @ ensures used==0; */
24   public void clear(){
25     used=0;
26     size=data.length;
27   }
28
29   /*@ normal_behavior
30     @ requires !this.full();
31     @ ensures used == \old(used)+1 && data[\old(used)] == b; */
32   public void put(byte b){
33     data[used]=b;
34     used = used + 1;
35   }
36
37   public static void main(String args[]){
38     Buffer buffer=new Buffer(512);
39     buffer.clear();
40     while(!buffer.full()){
41       buffer.put((byte)37);
42     }
43   }
44 }
```

Exercise 6: Static Checking (15 points)

(a) *(5 points)* Consider the method "find" given below, which checks if a given value "val" occurs in a given array "data". Provide a contract and a loop invariant for "find" that allows the JML static checker to prove the correctness of "find".

```
1 public boolean find(int data[], int val) {
2         boolean res = false;
3         int k = 0;
4         while (k < data.length) {
5                 if (data[k] == val) { res = true; }
6                 k = k + 1;
7         }
8         return res;
9 }
```

(b) *(6 points)* Consider the class `AgeStatistics`. When using the static checker to validate this class, two methods will be marked as containing an error. Which methods and what errors will be complained about?

```
1  public class AgeStatistics {
2    /*@ spec_public @*/ private int nr_of_children = 0;
3    /*@ spec_public @*/ private int nr_of_adults = 0;
4
5    /** This method can be used to record the birth or immigration
6        of children. */
7    /*@ requires count>=0;
8      @ ensures  nr_of_children == \old(nr_of_children) + count;
9      @ modifies nr_of_children; @*/
10   public void add_new_borns(int count){
11     nr_of_children=nr_of_children+count;
12   }
13
14   /** This method can be used to record when children
15       reach adulthood. */
16   /*@ requires 0 <= count && count <= nr_of_children;
17     @ ensures nr_of_children + nr_of_adults ==
18                \old(nr_of_children + nr_of_adults);
19     @ ensures nr_of_adults == \old(nr_of_adults) + count;
20     @ modifies nr_of_adults; @*/
21   public void record_new_adults(int count){
22     nr_of_children=nr_of_children-count;
23     nr_of_adults = nr_of_adults + count;
24   }
25
26   /** This method can be used to record the immigration of
27       adults. */
28   /*@ requires count>=0;
29     @ ensures  nr_of_adults == \old(nr_of_adults) + count;
30     @ modifies nr_of_adults; @*/
31   public void add_new_adults(int count){
```

```
32        nr_of_adults = nr_of_adults + count;
33     }
34
35     /** This method can be used to record the emigration of
36         families. */
37     /*@ requires 0 <= children && children <= nr_of_children;
38       @ requires 0 <= adults && adults <= nr_of_adults;
39       @ ensures  nr_of_children ==
40                  \old(nr_of_children) - children;
41       @ ensures nr_of_adults == \old(nr_of_adults) - adults;
42       @ modifies nr_of_children, nr_of_adults; @*/
43     public void record_emigration(int children, int adults){
44       add_new_borns(-children);
45       add_new_adults(-adults);
46     }
```

(c) *(4 points)* Consider the following extension to the class `AgeStatistics`. When validating the extension class, the static checker will signal one error. What is this error, and what causes it?

```
1    /** This method can be used to record when adults die or
2        emigrate. */
3    /*@ requires 0 <= count && count <= nr_of_adults;
4      @ ensures  nr_of_adults == \old(nr_of_adults) - count;
5      @ modifies nr_of_adults, nr_of_children; @*/
6    public void remove_adults(int count){
7      nr_of_adults = nr_of_adults - count;
8    }
9
10   /** This method can be used to record births and deaths. */
11   /*@ requires 0 <= deaths && deaths <= nr_of_adults;
12     @ requires births >= 0;
13     @ ensures  nr_of_adults == \old(nr_of_adults) - deaths;
14     @ ensures  nr_of_children == \old(nr_of_children) + births;
15     @ modifies nr_of_adults, nr_of_children; @*/
16   public void record(int births, int deaths){
17     add_new_borns(births);
18     remove_adults(deaths);
19   }
```

Exercise 7: Modelling (20 points)

In this exercise we model a door control system:

- During office hours, the door automatically opens when a person stands in front, and automatically closes when the person got through.
- During non-office hours, the door only opens when a valid employee card is presented.

We model the control system by defining three modules: *(i)* the *environment* that changes between office hours and non-office hours and determines the arrival of people; *(ii)* the *door system* itself; and *(iii)* a *main* module that connects the environment and the door.

(a) (*5 points*) Model the environment as a NuSMV module.

(b) (*6 points*) Model the door control system as a NuSMV module. Also define the main module that connects the environment and the door.

(c) (*3 points*) How can it be ensured that if a person has a valid card, he eventually can enter?

(d) (*3 points*) Specify the following property: After working hours, no person can enter without a valid card.

(e) (*3 points*) Specify the following property: When a person wants to enter, there always is a possibility that he/she will enter.