EXAM
System Validation
(192140122)
13:45 - 16:45
22-01-2015

- 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 a 4).
- The exam is open book: all *paper* copies of slides, papers, notes etc. are allowed.

Exercise 1: Formal Tools and Techniques (10 points)

The company that you work for makes a variety of small electric vehicles. They have decided to start working on autonomous vehicles and won the bid to provide vehicles for a big outdoor exhibition, to be held in the summer next year. These vehicles must act as both transportation and tour guides. That is, they must drive visitors around safely and while driving around explain the exhibits to the visitor, while following the instructions of the passenger on where to go next.

Knowing that errors tend to show up when the unexpected happens, a big emergency stop-and-reset button has been designed in and the system resets very quickly.

The team that is responsible for testing the main control software that looks for input from the sensors and provides commands to the subsystems is a broad mix of engineers, with both very theoretical and practical skills. They have asked you for a recommendation on which formal methods to use for which part of the control software.

Give a short description (no more than 200–250 words) of your advice to this team.

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*) Given a system whose state is either idle, or active, or completed.

Specify that the only two legal changes of state are from idle to active and from active to completed.

(b) (*3 points*) The variables `x`, `y`, and `z` never have the same value.

(c) (*4 points*) Given a Person `p`. This person regularly visits a bar, where `p` will stay until `p` is drunk and then leave.

Exercise 3: Modeling (15 points)

Consider a simple model of a battery system that consists of three battery cells and a battery controller below. The controller simply switches from one battery to the next in every round.

(a) (*9 points*) Write the remainder of the cell module to model a battery cell whose charge changes according to the following rules:

- If the cell is on-line and non-empty the charge decreases by 1 in every step.
- If the cell is off-line for two periods then the the charge is increased by 1, unless the battery was dead (empty).

(b) Write specifications for the following properties:

- (*3 points*) At any time precisely one battery is on-line.
- (*3 points*) If battery c1 is on-line and has reached charge level 2 or less, then the controller must make a non-deterministic choice between the other two batteries.

```
1 MODULE cell
2
3 VAR
4   online : boolean;
5   charge : 0..8;
6
7  ...
8
9 MODULE controller(c1,c2,c3)
10
11 ASSIGN
12   init(c1.online):=TRUE;
13   init(c2.online):=FALSE;
14   init(c3.online):=FALSE;
15   next(c1.online):=c3.online;
16   next(c2.online):=c1.online;
17   next(c3.online):=c2.online;
18
19 MODULE main
20
21 VAR
22    c1   : cell;
23    c2   : cell;
24    c3   : cell;
25    ctrl : controller(c1,c2,c3);
```

Exercise 4: Abstraction (10 points)

Given are the following code fragments of a Java package dealing with a payment system.

```
1 public interface Token {
2
3   public String name();
4 }
```

```
1 public class GoldPiece implements Token {
2
3   private String name, head;
4
5   public String name(){
6     return name;
7   }
8
9   public String head(){
10    return head;
11  }
12 }
```

```
1 public class Diamond implements Token {
2
3   private String name;
4   private int carats;
5
6   public String name(){
7     return name;
8   }
9
10  public int weight(){
11    return carats;
12  }
13 }
```

```
1 public class Oracle {
2
3   public String prediction(Token payment){ /* implementation */ }
4 }
```

Write specifications for the interface Token and the classes GoldPiece, Diamond and Oracle (you can use the answer sheet on the last page of the exam), that express the following:

(a) Each token has a value.

(b) The value of a gold piece is fixed at 12 units.

(c) The value of a diamond is 7 units per carat.

(d) An oracle is given a token. The oracle can then give advice or not. However, if the value of the token is at least 10 units then the oracle always gives an answer.

Exercise 5: Software Model Checking (15 points)

In this exercise, we consider a program that models a lock. Specifically, the two vertical doors of the lock are modeled 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) (*6 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) (*6 points*) We will try to prove the given claim using the following two predicates:

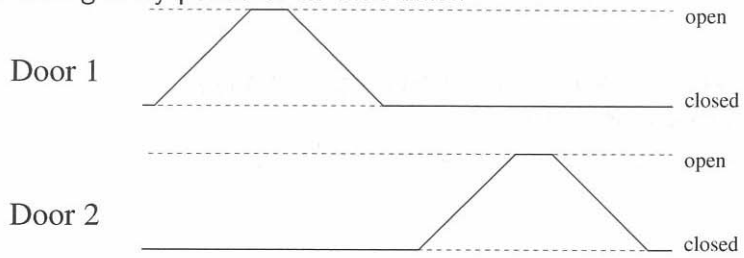
P1 door1==closed

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 6: Run-time Checking (15 points)

Each part is worth 5 points. Explain your answers (without explanation no points will be awarded).

(a) 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; this.y = y;
7    }
8
9    /*@ ensures \result <==> x > 0 && y > 0; */
10   public /*@ pure @*/ boolean firstQuadrant(){
11     return (x >= 0 && y >= 0);
12   }
13
14   public static void main(String args[]){
15     Pair p=new Pair(1,1);
16     boolean tmp=p.firstQuadrant();
17   }
18 }
```

(b) What will happen when the JML run-time checker is used to validate the following class? (See also exercise 7(a))

```
1  public class Find {
2    /*@ ensures (\forall int i;
3         0<=i && i<data.length ; data[i]!=val) ==> !\result;
4      @ ensures (\exists int i;
5         0<=i && i<data.length ; data[i]==val) ==> \result;
6      @*/
7    public static boolean find(int[] /*@non_null@*/ data, int val){
8      boolean res=false;
9      int k=0;
10     while(k<data.length){
11       if (data[k]==val) res=true;
12       k++;
13     }
14     return res;
15   }
16
17   public static void main(String args[]){
18     int data[]={3,31,5,65};
19     find(data,10);
20     find(data,31);
21   }
22 }
```

(c) 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     @*/
33   public void put(byte b){
34     data[used]=b;
35     used++;
36   }
37
38   public static void main(String args[]){
39     Buffer buffer=new Buffer(512);
40     buffer.clear();
41     while(!buffer.full()){
42       buffer.put((byte)37);
43     }
44   }
45 }
```

Exercise 7: Static Checking (15 points)

Subquestions (a) and (c) are worth 4 points, subquestion (b) is worth 7 points. Explain your answers (without explanation no points will be awarded).

(a) Consider the class `Find` from exercise 6(b) (page 6). Provide a loop invariant for the loop in the find method that will allow the JML static checker to prove the correctness of the find method.

(b) 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) 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 8: Test Generation with JML (10 points)

Consider an event logging facility for some system implemented in the class `Logger` below, where events are classes implementing the `Event` interface. The logger uses a cyclic buffer and stores only the last `MAX_LOG` events. Additionally the logger is "self-aware" that is, it also logs its own events of initialising and re-cycling the log buffer. The logger only logs non-null events and ones that are fresher than all the other events in the log (that is, the events have to arrive in a timely fashion and back logging is refused). Assume that the total number of events is never going to be an issue in terms of overflow, that is, the system is short-lived enough not to produce more than 2^{31} events allowed by the `int` data type for `totalEvents`.

(a) (*5 points*) Provide **one** complete as you can JML specification case that would be sufficient to thorougly test the most common execution path of the `log` method, that is one where a correct even is passed to it and the log does not need to be initialised or re-cycled.

(b) (*2 points*) State how many other specification cases (without writing them down) you would need to provide to have a JML specification that would cause the test generator to test the `log` method to a sufficient degree. Provide argumentation for your answer. What else would you need to do for the test generator to provide useful test results?

(c) (*3 points*) The intended property of the log is that only fresher events (than the already existing in the log) are ever going to be added to the log. Assuming you can only look at the presented code and not the rest of the system, is it really the case? Explain your answer.

```
1  public interface Event { /*@ pure @*/ int getDate(); }
2
3  public class Logger {
4    private final Event[] log = new Event[1024];
5    private int totalEvents = 0, currentEntry = -1;
6
7    public void log(/*@ nullable @*/ Event event) {
8      if(event == null) throw new NullPointerException();
9      if(currentEntry >= 0 &&
10        log[currentEntry].getDate() >= event.getDate())
11       throw new IllegalArgumentException();
12     if(currentEntry < 0) {
13       currentEntry++;
14       log[currentEntry] = new LogInitEvent();
15       totalEvents++;
16     }
17     .currentEntry++;
18     if(currentEntry == log.length) {
19       currentEntry = 0;
20       totalEvents++;
21       log[currentEntry] = new LogRecycleEvent();
22       currentEntry++;
23     }
24     totalEvents++;
25     log[currentEntry] = event;
26   }
27 }
```