EXAM
System Validation
(192140122)
08:45 - 11:45
06-11-2017

- 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 (10 points)

The Japanese railways are developing a new version of their Shinkansen high speed train. Just as the existing Shinkansen trains, it is essential that the trains are always on time. A new feature is that a personal entertainment system is added on board of the train. The software for the personal entertainment system is added on top of the existing software that controls train speed and departure.

Suppose that you work as a quality engineer for the Japanese railways and you are asked to advice how to develop the software for this new train in the most efficient way. Write a short advice (max 250 words), in which you address the following points:

- How to ensure that the trains remain as punctual as before?
- What quality requirements would there be for the on-board personal entertainment system?
- How to ensure these quality requirements are met?

Exercise 2: Specification (15 points)

Write formal specifications for the following informal statements about Japan in an appropriate specification formalism of your choice. You may use different formalisms for the different subexercises. You may assume that appropriate atomic propositions, query methods, and classes exist.

(a) (*3 points*) Every day of the year, it may happen that you can see Mount Fuji.

(b) (*3 points*) The trains in Japan are never late.

(c) (*3 points*) When the train arrives, first the doors open, then passagers leave, and then new passagers enter.

(d) (*3 points*) You may only enter a temple, if you washed your hands, and took off your shoes.

(e) (*3 points*) Before the temple garden closes, all visitors should have left.

Exercise 3: Abstraction (15 points)

Suppose we have the following interface ChessPiece (where the type Color is defined as being black or white, and we have an auxiliary class Point). We assume that a chess board is implemented using 8x8 matrix with indices ranging from 0 to 7.

```
1 public enum Color {
2     black, white
3 }
```

```
1 class Point {
2     /*@ spec_public */ private int x;
3     /*@ spec_public */ private int y;
4
5     Point(int x, int y) {
6         this.x = x;
7         this.y = y;
8     }
9
10    //@ ensures \result == x;
11    public int getX() {
12        return x;
13    }
14
15    //@ ensures \result == y;
16    public int getY() {
17        return y;
18    }
19 }
```

```
1 import java.util.List;
2
3 public interface ChessPiece {
4
5     public Color getColor();
6
7     public Point getCurrentPosition();
8
9     public List<Point> getNextPositions();
10 }
```

(a) *(7 points)* Write a JML specification for the interface ChessPiece using model variables for color, position and possible next positions. Add method contracts in terms of these model variables, and add specifications to express that the following properties always hold:

- pieces are black or white
- pieces are on the 8x8 board,
- possible next positions are all the positions on the board that a piece can move to, and which are different from the current position (N.B., this method only considers reachability, not whether a field is empty or not).

(b) *(3 points)* Consider the implementation of AbstractChessPiece, which implements the joint functionality of all chess pieces (i.e., color and current position), but does not implement the computation of the possible next positions.

```
1  import java.util.List;
2
3  public abstract class AbstractChessPiece implements ChessPiece{
4
5      private Color color;
6
7      private Point position;
8
9      public Color getColor() {
10         return color;
11     }
12
13     public Point getCurrentPosition() {
14         return position;
15     }
16
17     public abstract List<Point> getNextPositions();
18
19 }
```

Give sufficient specifications to show that this correctly implements the interface ChessPiece as you specified it.

(c) *(5 points)* Consider the implementation of a rook in class Rook.

```
1  import java.util.*;
2
3  public class Rook extends AbstractChessPiece {
4
5      public List<Point> getNextPositions() {
6          List<Point> res = new ArrayList<Point>();
7          Point p = getCurrentPosition();
8          int x = p.getX();
9          int y = p.getY();
10         for (int i = 0; i < 8; i++) {
11             if (i != x) {
12                 res.add(new Point(i, y));
13             }
14             if (i != y) {
15                 res.add(new Point(x, i));
16             }
17         }
18         return res;
19     }
20 }
```

Provide sufficient specifications to show that this correctly implements the interface ChessPiece as you specified it.

Exercise 4: Static Checking (15 points)

Consider the implementation of a class ChessBoard (which uses the ChessPiece interface of the previous exercise).

```
1  class ChessBoard {
2
3      ChessPiece [][] board = new ChessPiece [8][8];
4
5      /*@ ensures \result ==
6              (\forall int i; 0 <= i && i < 8;
7              (\forall int j; 0 <= j && j < 8;
8                  (board[i][j] == null ==> 2 <= i && i <=5) &&
9                  (board[i][j] != null &
10                  board[i][j].getColor() == Color.white ==>
11                      i == 0 || i == 1) &&
12                 (board[i][j] != null &
13                  board[i][j].getColor() == Color.black ==>
14                    i == 6 || i == 7)));
15      */
16     public boolean checkInitialState() {
17         boolean okay = true;
18         for (int i = 0; i < 8; i++) {
19             okay = okay && board[0][i] != null &&
20                     board[0][i].getColor() == Color.white &&
21                     board[1][i] != null &&
22                     board[1][i].getColor() == Color.white;
23         }
24         for (int i = 0; i < 8; i++) {
25             okay = okay && board[6][i] != null &&
26                     board[6][i].getColor() == Color.black &&
27                     board[7][i] != null &&
28                     board[7][i].getColor() == Color.black;
29         }
30         for (int i = 0; i < 8; i++) {
31             for (int j = 2; j < 6; j++) {
32                 okay = okay && board[j][i] == null;
33             }
34         }
35         return okay;
36     }
37
38
39
40
41
42
43
44
45
```

```
46      /*@ ensures (\forall int i; 0 <= i && i <= 7;
47                   (\forall int j; 0 <= j && j <= 7;
48                   (board[i][j] == c && p.getX() == i && p.getY() == j) ||
49                   (board[i][j] == \old(board[i][j])))); */
50      public void moveTo(ChessPiece c, Point p) {
51          Point curPos = c.getCurrentPosition();
52          if (legalMove(c, p)) {
53              board[curPos.getX()][curPos.getY()] = null;
54              board[p.getX()][p.getY()] = c;
55          }
56      }
57
58      /*@ assignable \everything;
59        ensures true; */
60      public boolean legalMove(ChessPiece c, Point nextPos) {
61          return c.getNextPositions().contains(nextPos) &&
62              noConflicts(c, nextPos);
63      }
64
65      /*@ pure */ public boolean noConflicts(ChessPiece c, Point nextPos) {
66          // stub implementation, should be replaced by a real check
67          return true;
68      }
69  }
```

(a) (*6 points*) The method checkInitialState checks if the board is in the initial state, i.e. the first two rows are filled with white pieces, the last two rows are filled with black pieces, and all rows in between are empty.

Provide loop invariants for all loops in this method (it is sufficient to give a loop invariant for the outer loop in the last loop). Hint: you can use the keyword \pre(E) to denote the value of the expression E in the pre-state of the loop.

(b) (*4 points*) The interface ChessPiece defines a method getCurrentPosition(). Add an invariant to the class ChessBoard that captures that all pieces know that they are at the correct position.

(c) (*5 points*) Consider the methods moveTo(ChessPiece c, Point p) and legalMove(ChessPiece c, Point p). The legalMove method uses a method noConflicts, which implements the precise chess rule, and that we do not consider further here. Will static verification of method moveTo succeed? Motivate your answer. (Without motivation, no points will be awarded.)

Exercise 5: Run-time Checking (15 points)

Explain your answers (without explanation no points will be awarded). Consider the class Server.

```
1  import java.util.*;
2
3  public class Server {
4
5      List<Server> connections;
6      public static final int maxConnections = 8;
7
8      //@ invariant connections.size() <= maxConnections;
9      //@ invariant 1 <= connections.size();
10
11     public List<Server> getConnections() {
12         return connections;
13     }
14
15     //@ ensures getConnections().contains(s);
16     public void connect(Server s) {
17         if (connections.size() < maxConnections) {
18             connections.add(s);
19         }
20     }
21
22     /*@ requires getConnections().size() + newConnections.size() <
23                                                          maxConnections;
24         ensures (\forall Server s;
25                  newConnections.contains(s); getConnections().contains(s));
26     */
27     public void massConnections(List<Server> newConnections) {
28         connections.addAll(newConnections);
29     }
30
31     public Server() {
32         connections = new ArrayList<Server>();
33     }
34
35     public static void main (String [] args) {
36         Server s1 = new Server();
37         Server s2 = new Server();
38         s1.connect(s2);
39         List<Server> l = new ArrayList<Server>();
40         for (int i = 0; i < maxConnections; i++) {
41             l.add(new Server());
42         }
43         s1.massConnections(l);
44     }
45 }
```

(a) (*10 points*) When the runtime checker is used on this class, two different problems will be detected. What are these problems, and why are they happening?

(b) (*5 points*) The postcondition of the method connect is wrong, but this will not be detected by this main method. Explain what you could do to make the runtime checker detect this problem.

Exercise 6: Abstraction-refinement (10 points)

Consider the model in Figure **??**, with the state labeled s_1 as the initial state.

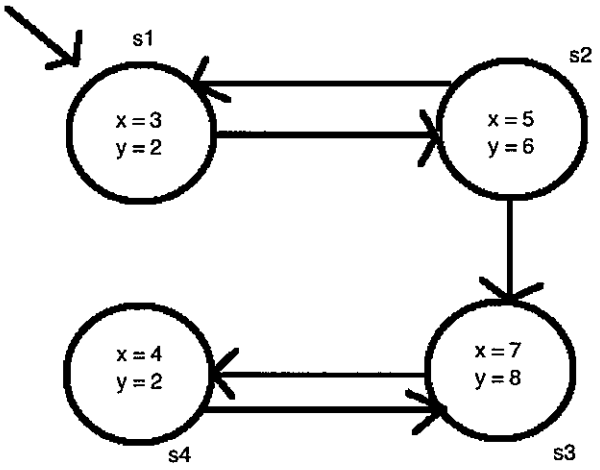

Figure 1: Concrete Model

Suppose we have the following predicates: even_x and even_y, formally defined as:

$$\begin{aligned} \text{even}_x &\triangleq x\%2 = 0 \\ \text{even}_y &\triangleq y\%2 = 0 \end{aligned}$$

(a) (4 points) Construct an abstract model from the concrete model in Figure **??** w.r.t. the predicates even_x and even_y.

(b) (3 points) What happens if you try to check the property "X $\neg\text{even}_x$" (*i.e.* in a state that is reachable in one step, the value of x is not even) over the abstract model? What can you conclude about the concrete model from this result? Motivate your answer.

(c) (3 points) What happens if you try to check the property "G even_y" (*i.e.* the value of y is always even) over the abstract model? What can you conclude about the concrete model from this result? Motivate your answer.

Exercise 7: Modelling (20 points)

In this exercise we model a Japanse toilet system, which has functionality to heat the toilet seat, a bidet function, and a function to make flushing sounds.

Instructions for a typical Japanse toilet seat system.
This is not the system modelled in this exercise.

- The system has buttons which can be pressed by the user to:
  - switch the heating on or off;
  - to make a request for the bidet function; and
  - to make a request to switch on the flushing sounds.
- Heating can always be switched on and off.
- The bidet function, and the function to make flushing sounds cannot be switched on simultaneously. However, the request to switch the function on is remembered.
- Both the bidet function and the flushing sounds are switched off automatically after 10 steps.
- If the bidet or the flushing sound function switches off, first a pending bidet request is handled, then a pending flushing sound request.
- It is allowed to make a new request for a function that is currently activated.

(a) (*10 points*) Model this system as a NuSMV module.

(b) (*4 points*) Model a user module that randomly presses buttons and a main module that composes the toilet and the user module.

(c) (*3 points*) Specify the following property: the bidet and the flushing sounds are never switched on simultaneously.

(d) (*3 points*) Specify the following property: It is always possible to switch on the toilet seat heating.