EXAM
System Validation
(192140122)
13:45 - 17:15
04-11-2013

- 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)

Consider an on-line shopping environment, consisting of several components such as:

- a payment component;
- a display results component that shows customers the items that correspond to their search criteria;
- a suggestion component that shows customers the items that are probably of interest for the customers, given their earlier searches and purchases; and
- an order component, taking care that once the payment is in, the item is shipped.

Recently, there have been several complaints from customers that they ordered some item, and never received it. On various web forums, customers also complain about the strange results displayed when they are searching for some particular item.

Within the company, the management has decided that serious damage control is needed, and that the software needs to be analysed thoroughly and improved to avoid such problems happening again in the future. The improvements in the system should be integrated as fast as possible.

You have recently been hired by the company as the software quality officer, and therefore you are asked to make a plan that will ensure that the reputation of the company is not further damaged. Your plan should provide a balance between effort, money and time needed, and the potential reputation damage to the company.

Describe your plan in at most 200 words. You may assume that different formal validation tools exist for the programming language that has been used for this system.

Answer

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

- Payment and shipping system should be 100% correct: if a customer pays, he should get the item delivered.

- Displaying of results should be tested (if possible with annotations added, to find more errors), taking cases on forum as a starting point. It should improve, but if display result is not always optimal, consequences will not be as serious as when the payment or order component makes a mistake.
- Time is too short to establish full correctness of whole application.
- Collaboration between payment and ordering component maybe via satabs – control flow properties?

Exercise 2: Specification (12 points)

Write formal specifications for the following informal requirements in an appropriate specification formalism of your choice (3 points per item). You may assume that appropriate atomic propositions, query methods and classes exist.

a When you upload your assignment in Blackboard, it will be graded.

b Every weekend, the heating system is switched off, until the water temperature gets below 15 degrees.

c There always is a Dutch skater that holds a world record.

d The number of people in the room is never in- or decreased by more than 1.

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 G $(uploaded \rightarrow \mathsf{F}\ graded)$

b $G(weekend \rightarrow (heating\_off\ \mathsf{W}\ temp\_below\_15))$

c `invariant (\exists Skater s.  dutch(s) && holds_worldRecord(s));`

d `constraint Math.abs(\old(nr_people) - nr_people) <= 1;`

Exercise 3: Conveyor Belt Modelling (18 points)

Consider a warehouse where items are loaded onto a truck, by means of a conveyor belt that runs between the warehouse and the truck. The conveyor belt is not 100% reliable and items may fall off the belt. If that happens an alarm must be raised and everything must stop until the item is placed back onto the belt.

A number of events related to items are possible:

- The warehouse is restocked with items.
- An item leaves the warehouse on the belt.
- An item falls off the belt.
- An item is put back onto the belt.
- An item reaches the truck.
- A truck full of items leaves and is replaced by an empty one.

a (*9 points*) Write an SMV model of this system, that models the behaviour of the items and the alarm:

- The items shall not be modeled as separate modules, but as a count of items in the warehouse, on the belt, and in the waiting truck.
- In every step in the model, at most one item-related event can happen.
- The model shall include the behaviour of the alarm.

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.

In the answers to exercises b–d, you can use the variables containing the item counts and/or the state of the alarm. Moreover, for every integer variable ending in _count, such as truck_count, you may assume that a variable truck_diff exists, which is 0 in the initial state and denotes the difference between the current and previous count value in all later states. Assume that these have been automatically defined for you as:

```
1 init(truck_diff):=0;
2 next(truck_diff):=next(truck_count)-truck_count;
```

Using auxiliary variables is permitted if you DEFINE them in terms of the allowed ones.

b (*3 points*) Specify that every truck is eventually fully loaded and leaves.

c (*3 points*) Specify that it is possible that a truck is loaded without any items falling of the belt.

d (*3 points*) We wish to ensure that the model contains no runs in which items keep falling off the belt and being put back, without any items ever reaching the truck. Add some information to the model that ensures this.

Answers

```
1
2 MODULE main
3
4 DEFINE
```

```
5    WC := 100; -- warehouse_count capacity
6    BC := 5  ; -- belt_count capacity;
7    TC := 10 ; -- truck_count capacity
8
9  VAR
10     event : {restock,leave,fall,replace,enter,depart};
11     warehouse_count : 0..WC;
12     belt_count : 0..BC;
13     truck_count : 0..TC;
14     alarm : boolean;
15
16 ASSIGN
17     init(warehouse_count):=WC;
18     init(belt_count):=0;
19     init(truck_count):=0;
20     init(alarm):=FALSE;
21     next(warehouse_count):=case
22         event=restock : WC;
23         event=leave   &warehouse_count>0&belt_count<BC : warehouse_count - 1;
24         TRUE : warehouse_count;
25     esac;
26     next(belt_count):=case
27         event=leave   &warehouse_count>0&belt_count<BC : belt_count+1;
28         event=fall    &belt_count>0 : belt_count - 1;
29         event=replace &belt_count<BC : belt_count+1;
30         event=enter   &belt_count>0&truck_count<TC : belt_count - 1;
31         TRUE : belt_count;
32     esac;
33     next(truck_count):=case
34         event=enter   &belt_count>0&truck_count<TC : truck_count+1;
35         event=depart  &truck_count=TC : 0;
36         TRUE:truck_count;
37     esac;
38     next(alarm):=case
39         event=fall    &belt_count>0 : TRUE;
40         event=replace &belt_count<BC : FALSE;
41         TRUE:alarm;
42     esac;
43
44 -- define differences using monitors.
45
46 VAR
47     warehouse_diff : -WC..WC;
48     belt_diff : -BC..BC;
49     truck_diff:-TC..TC;
50
51 ASSIGN
52     init(warehouse_diff):=0;
53     next(warehouse_diff):=next(warehouse_count) - warehouse_count;
```

```
54      init(belt_diff):=0;
55      next(belt_diff):=next(belt_count) - belt_count;
56      init(truck_diff):=0;
57      next(truck_diff):=next(truck_count) - truck_count;
58
59 LTLSPEC G F truck_diff=-TC;
60
61 CTLSPEC EF (truck_count=0 & E[( belt_diff=-1 -> truck_diff=1) U truck_count=TC])
62
63 FAIRNESS truck_diff>0;
```

Exercise 4: Software Model Checking (12 points)

Consider the following program

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

This program has already been annotated to show that precisely one of the lights will be on after an update.

a (*4 points*) Without knowing how the file update.c implements the update procedure, annotate the main body to verify that this traffic light runs in a sequence:

$$\text{red} \rightarrow \text{green} \rightarrow \text{yellow} \rightarrow \text{red} \rightarrow \cdots$$

Note that update is not required to change the light on every call.
Hint: use an auxiliary variable to keep track of the color of the traffic light before the call to update.

b (*6 points*) We will try to prove the given claim using the following two predicates:

P0 red+yellow+green==0

P1 red+yellow+green==1

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

```
boolean P0,P1=F,T;
for (;;){
    update();
    assert(P1);     // assert(red+yellow+green==1);
    P0,P1 := P0,P1; // sleep(1);
    P0,P1 := P0,P1; // counter=counter+1;
}
```

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

```
1 void update(){
2   red=0;
3   yellow=0;
4   green=0;
5   if ((counter%16) >= 1 && (counter%16) < 4) {
6     green=1;
7   } else if ((counter%16) >= 4 && (counter%16) < 6) {
8     yellow=1;
9   } else {
10    red=1;
11  }
12 }
```

c (*2 points*) Why is the given abstraction not good enough to prove that the assertion in the main program never fails?

Answers

a
```
1 unsigned int red=1;
2 unsigned int yellow=0;
3 unsigned int green=0;
4 unsigned int counter=0;
5
6 #include "update.c"
7
8 int main(int argc,char*argv[]){
9   for (;;){
10    int state=red?1:(green?2:3);
11    update();
12    assert(state==1 && (red || green)
13         ||state==2 && (green || yellow)
14         ||state==3 && (yellow || red) );
15    assert(red+yellow+green==1);
16    sleep(1);
17    counter=counter+1;
18  }
19 }
```

b
```
update(){
    P0,P1 := P0?T:*,*; // red=0;
    P0,P1 := P0?T:*,*; // yellow=0;
    P0,P1 := P0?T:*,*; // green=0;
    if (*) { //((counter%16) >= 1 && (counter%16) < 4) {
      P0,P1 := F,P0?:T:*; //  green=1;
    }else if (*) {//((counter%16) >= 4 && (counter%16) < 6) {
      P0,P1 := F,P0?:T:*; //  yellow=1;
    } else {
      P0,P1 := F,P0?:T:*; //  red=1;
    }
  }
```

c The three assignments at the start of the update block have to be translated separately. Only when they are treated as an atomic block can you conclude that if P1 is true before then P0 is true afterwards.

Exercise 5: Abstraction (12 points)

Consider the interface `Item` representing an item kept in a `BookStore` database quoted on the next page. The store keeps track of the type of the item for two purposes: (1) to calculate package and postage costs for on-line shoppers, and (2) to maintain a certain minimal state of the item supply for some types of items.

a (4 points) Annotate the `Item` class with class invariants and an abstract specification to distinguish between paper-printed media and digital media. (Assume computer games are sold as digital media).

b (4 points) Using the developed abstraction provide a concise specification for the `getWeight()` method in the `Item` class, stating which items *do* have a weight and which items *do not* have a weight.

c (4 points) Add appropriate specifications to the `BookStore` class to state that there are always books and newspapers in stock. Provide two alternatives of this specification, one using the `getCount()` method, and one not using it.

```java
public interface Item {

        public final static int BOOK = 1;
        public final static int NEWSPAPER = 2;
        public final static int MAGAZINE = 2;
        public final static int BOARD_GAME = 3;
        public final static int DIGITAL_BOOK = 4;
        public final static int COMPUTER_GAME = 5;
        public final static int AUDIO_BOOK = 6;

        public int getType();

        public int getWeight();
}
```

```java
import java.util.ArrayList;
import java.util.List;

public class BookStore {

        private List<Item> store = new ArrayList<Item>();

        public int getCount(int type) {
                int count = 0;
                for(Item i : store) {
                        if(i.getType() == type) {
                                count++;
                        }
                }
                return count;
        }

        public int sell(Item item) {
                store.remove(item);
                return item.getWeight();
        }

        public void newArrival(Item item) {
                store.add(item);
        }
}
```

Answer

```
1  public interface Item {
2
3          public final static int BOOK = 1;
4          public final static int NEWSPAPER = 2;
5          public final static int MAGAZINE = 2;
6          public final static int BOARD_GAME = 3;
7          public final static int DIGITAL_BOOK = 4;
8          public final static int COMPUTER_GAME = 5;
9          public final static int AUDIO_BOOK = 6;
10
11         /*@ public instance invariant
12           @   getType() == BOOK || getType() == NEWSPAPER ||
13           @   getType() == MAGAZINE || getType() == BOARD_GAME ||
14           @   getType() == DIGITAL_BOOK || getType() == COMPUTER_GAME ||
15           @   getType() == AUDIO_BOOK; @*/
16
17         //@ public instance model boolean digital;
18         /*@ public represents digital =
19           @    (getType() == DIGITAL_BOOK ||
20           @     getType() == COMPUTER_GAME ||
21           @     getType() == AUDIO_BOOK); @*/
22
23         //@ public instance model boolean paper;
24         //@ public represents paper = !digital;
25
26         public /*@ pure @*/ int getType();
27
28         //@ ensures digital ==> \result == 0;
29         //@ ensures paper ==> \result > 0;
30         public /*@ pure @*/ int getWeight();
31 }
```

```
1  import java.util.ArrayList;
2  import java.util.List;
3
4  public class BookStore {
5
6          private /*@ spec_public @*/ List<Item> store
7                  = new ArrayList<Item>();
8
9          //@ public instance invariant getCount(Item.BOOK) > 0;
10         //@ public instance invariant getCount(Item.NEWSPAPER) > 0;
11
12         public /*@ pure @*/ int getCount(int type) {
13                 int count = 0;
14                 for(Item i : store) {
15                         if(i.getType() == type) {
16                                 count++;
```

```
17                          }
18                  }
19                  return count;
20          }
21
22          // Alternatively not using getCount():
23          /*@ public instance invariant
24            @   (\exists int i;
25            @       i>=0 && i < store.size();
26            @       store.get(i).getType() == Item.BOOK);
27            @ public instance invariant
28            @   (\exists int i;
29            @       i>=0 && i < store.size();
30            @       store.get(i).getType() == Item.NEWSPAPER);
31            @*/
32
33          /**
34           * @param item item being sold
35           * @return the weight of the item to calculate p&p.
36           */
37          public int sell(Item item) {
38                  store.remove(item);
39                  return item.getWeight();
40          }
41
42          public void newArrival(Item item) {
43                  store.add(item);
44          }
45  }
```

Exercise 6: Run-time Checking (12 points)

Explain your answers (without explanation no points will be awarded). If you think the specification is violated, clearly indicate at what point in the execution, the problem will occur.

a (6 points) Consider class `BinarySearchRecursive`. What will happen when the JML run-time checker is used to validate this class? (You can assume that RAC can correctly check the validity of the quantifier in the postcondition).

```
1  public class BinarySearchRecursive {
2
3      //@ requires l > 0 && s >= 0 && s + l <= a.length;
4      //@ requiers (\forall int i; 0<i && i<a.length; a[i-1] < a[i]);
5      /*@ ensures \result <==>
6        @   (\exists int i; i>=0 && i<l; a[s+i] == t); @*/
7      public static boolean search(int s, int l, int[] a, int t) {
8          if(l == 1) return (a[s] == t);
9          int m = s + l/2;
10         if(a[m] == t) return true;
11         if(a[m] > t) return search(s, l/2, a, t);
12         return search(m, l/2, a, t);
13     }
14
15     public static void main(String[] args) {
16         int[] a1 = {1, 2, 3, 4, 5, 6, 7, 8};
17         search(0, a1.length, a1, 3);
18         search(0, a1.length, a1, 0);
19     }
20 }
```

b (2 points) If you have found a problem in the previous point, change the `main` method to reveal the problem. If you have not found any problems and you think there is none, argue why.

c (4 points) Consider class `TransactionRegister` on the next page. What will happen when the JML run-time checker is used to validate this class?

```
1 public class TransactionRegister {
2
3     private /*@ spec_public @*/ short transactionCount = 0;
4
5     //@ public constraint \old(transactionCount) <= transactionCount;
6     //@ public invariant transactionCount <= 100000;
7
8     /** Registers a transaction if the maximum number of
9         transactions has not yet been reached. */
10    void confirmTransaction() {
11        if(transactionCount <= 100000) transactionCount++;
12    }
13
14    public static void main(String[] args) {
15        TransactionRegister trs = new TransactionRegister();
16        for(int i=0; i < Integer.MAX_VALUE; i++) {
17            trs.confirmTransaction();
18        }
19    }
20 }
```

Answers

a Nothing will happen, because the length of the input array is a power of 2. For arrays that are not an even power of 2 in length, some elements will be missed out during the search.

b

```
1         public static void main(String[] args) {
2                 int[] a2 = {1, 2, 3, 4, 5, 6};
3                 search(0, a2.length, a2, 3 /* or 6 */);
4         }
```

c The RAC will fail. This is because there is a subtle data-type bug, `transactionCount` is a `short` integer, once it reaches 32767 it will leap back to the minimum negative number for the short -32768 invalidating the constraint. Note that the class invariant is still going to be satisfied.

Exercise 7: Static Checking (12 points)

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

a (4 points) Consider class `Matrix`. Provide a class invariant to specify that the matrix is square in dimension, and a complete loop specification so that the postcondition of the method `resetDiagonal` can be established.

```
1  public class Matrix {
2
3          private /*@ spec_public @*/ int[][] contents;
4
5          private /*@ spec_public @*/ int size;
6          //@ public invariant size > 0;
7
8          //@ ensures (\forall int i; i>=0 && i<size; contents[i][i]==0);
9          public void resetDiagonal() {
10                 for(int i=0; i<size; i++) {
11                         contents[i][i] = 0;
12                 }
13         }
14 }
```

b (4 points) Consider class `Rectangle` on the next page. What will happen when Extended Static Checker is used to validate this class?

c (4 points) Consider class `PIN` on the next page. What will happen when Extended Static Checker is used to validate this class?

```
public class Rectangle {
        private /*@ spec_public @*/ int width;
        private /*@ spec_public @*/ int height;

        //@ ensures width == \old(width) * factor;
        //@ assignable this.*;
        public void scaleX(int factor) {
                width = width * factor;
        }

        //@ ensures height == \old(height) * factor;
        //@ assignable this.*;
        public void scaleY(int factor) {
                height = height * factor;
        }

        //@ ensures width == \old(width) * factor;
        //@ ensures height == \old(height) * factor;
        //@ assignable this.*;
        public void scaleBoth(int factor) {
                scaleX(factor);
                scaleY(factor);
        }
}
```

```
public class PIN {
        private /*@ spec_public @*/ int pin, tryCounter;
        //@ public invariant tryCounter >= 0 && tryCounter <= 3;
        private /*@ spec_public @*/ boolean access;

        //@ requires pin == this.pin;
        //@ ensures \result ==> this.tryCounter == 3;
        public boolean checkPin(int pin) {
                boolean result = false;
                if(pin == this.pin && tryCounter > 0) {
                        result = true;
                        resetCounter();
                }else if(tryCounter > 0) tryCounter--;
                this.access = result;
            return result;
        }

        //@ requires access;
        //@ ensures tryCounter == 3;
        //@ assignable tryCounter;
        public void resetCounter() {
                if(access) tryCounter = 3;
        }
}
```

Answers

a

```
1 public class Matrix {
2
3         private /*@ spec_public @*/ int[][] contents;
4         //@ public invariant contents.length == size;
5         /*@ public invariant (\forall int i; i>=0 && i<contents.length;
6               contents[i] != null && contents[i].length == size); @*/
7         private /*@ spec_public @*/ int size;
8         //@ public invariant size > 0;
9
10        //@ ensures (\forall int i; i>=0 && i<size; contents[i][i]==0);
11        public void resetDiagonal() {
12                /*@ loop_invariant i>=0 && i<=size;
13                  @ loop_invariant
14                    (\forall int k; k>=0 && k < i; contents[k][k] == 0);
15                  @ decreases contents.length - i;
16                  @*/
17                for(int i=0; i<size; i++) {
18                        contents[i][i] = 0;
19                }
20        }
21 }
```

b The `assignable` condition of both `scaleX` and `scaleY` methods is too wide, for each of the methods the corresponding postcondition only states what happens to one of the dimensions of the rectangle, while the assignable clause states that both can change. ESC is not able to establish anything about the other dimension after the calls to `scaleX` and `scaleY` in the method `scaleBoth` and consequently the postcondition of `scaleBoth` cannot be established.

c The call to `resetCounter` in method `checkPin` does not satisfy the precondition, because the field `access` is updated only later in `checkPin`.

Exercise 8: Test Generation with JML (12 points)

Consider an upgraded version of the `SensorValue` class from the second homework assignment, with the code on the next page. This implementation has been upgraded to better deal with failing sensors. Instead of providing a fixed fail-safe value it provides a mean value of the last three correct readouts, while the fail-safe value is used when there were no three correct readouts yet. Additionally, it records and reports the type of failure (below minimum value, or above maximum value) and based on the number of failures sets the `permanent` flag to indicate that the failure is considered serious. You can disregard any overflow issues during the addition of integers.

a (6 points) Provide a JML contract specification for the method readSensor() that is going to produce enough meaningful tests to thoroughly test this method. This is an operation of a safety critical application, so every part of the state of the sensor counts. If you think that some part of this specification can be better stated as a class invariant, provide the class invariant instead.

b (3 points) Give an argument, as short as you can think of, why you have provided enough specification cases for this method.

c (3 points) When generating tests with JMLUnitNG one has to provide test data. With respect to providing this test data, try to describe what would you consider the most crucial factor to consider for this particular test generation scenario.

```
1  public class SensorValue {
2
3          public static final int STATUS_OK = 0,
4                                  STATUS_ERR_MIN = 1,
5                                  STATUS_ERR_MAX = 2;
6
7          private int value;
8          private final int failSafe, minValue, maxValue;
9          private int[] valueHistory = new int[3];
10         private int correctReadouts = 0, failedReadouts = 0;
11         private boolean permanent = false;
12
13         SensorValue(int failSafe, int minValue, int maxValue) {
14                 this.failSafe = failSafe;
15                 this.minValue = minValue;
16                 this.maxValue = maxValue;
17                 this.value = failSafe;
18         }
19
20         /** Register the newly read value, check for errors and report. */
21         public int readSensor(int newValue) {
22                 if(newValue < this.minValue || newValue > this.maxValue) {
23                         failedReadouts++;
24                         this.permanent = (this.failedReadouts >= 10);
25                         if(correctReadouts >= 3) {
26                                 this.value = (this.valueHistory[0] +
27                                               this.valueHistory[1] +
28                                               this.valueHistory[2])/3;
29                         }else{
30                                 this.value = this.failSafe;
31                         }
32                         if(newValue < this.minValue) {
33                                 return STATUS_ERR_MIN;
34                         }else{
35                                 return STATUS_ERR_MAX;
36                         }
37                 }else{
38                         this.value = newValue;
39                         correctReadouts++;
40                         this.valueHistory[2] = this.valueHistory[1];
41                         this.valueHistory[1] = this.valueHistory[0];
42                         this.valueHistory[0] = newValue;
43                         return STATUS_OK;
44                 }
45         }
46 }
```

Answer

a There is obviously more than one possible way to provide this specification, one has to balance the amount of specifications by occasionally merging specification cases:

```
1 //@ public invariant permanent <==> (failedReadouts >= 10);
```

```
1 /*@ public normal_behavior
2   requires minValue <= newValue && newValue <= maxValue;
3   ensures valueHistory[0] == value &&
4           valueHistory[1] == \old(valueHistory[0]) &&
5           valueHistory[2] == \old(valueHistory[1]);
6   ensures correctReadouts == \old(correctReadouts) + 1;
7 also public normal_behavior
8   requires minValue > newValue || newValue > maxValue;
9   ensures failedReadouts == \old(failedReadouts) + 1;
10  ensures correctReadouts >= 3 ==>
11    value == (valueHistory[0]+valueHistory[1]+valueHistory[2])/3;
12  ensures correctReadouts < 3 ==> value == failSafe;
13 also public normal_behavior
14  requires minValue > newValue;
15  ensures \result == STATUS_ERR_MIN;
16 also public normal_behavior
17  requires maxValue < newValue;
18  ensures \result == STATUS_ERR_MAX; @*/
```

b Every possible execution branch in the code has to be covered by at least one specification case.

c The difficulty in this particular case is that the object stores a history of values. Depending on this history the outcome of the method will be different. The test data generator has to invoke different sequences of `readSensor()` calls to build up the different possible states of the sensor value history.