EXAM
System Validation
(192140122)
13:45 - 17:15
05-11-2012

- 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 a company that sells hotel booking systems. Already several hotels have bought this system, and they are supposed to start working with this in a month from now.

The requirements for the system have been documented thoroughly, but informally. A first prototype of the system has been built, but the main developer of the system has been fired, because it turned out he was an unskilled programmer. Therefore, this first prototype is full with bugs, it is undocumented, and variable names have been chosen at random.

The company does not want to loose its reputation of providing good quality software. Therefore, to solve the problem, the requirements department is asked to identify as many bugs as possible, as quickly as possible. The system is too large and complex to do this with testing only (however, there is a skilled testing department that might be asked for support). Fortunately, all employees from the requirements department are trained users of formal methods.

Describe in at most 200 words how they could achieve this. You may assume that appropriate 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:

- Requirements are well-documented, should be formalised.
- Most likely more data-oriented, specify as JML-like annotations.
- Let the test department do testing with the annotations
- Static checking for selected parts (either because correctness is crucial, or because testers find annotation violations)
- Use SatABs (like) tool
- Time is too short to establish full correctness of whole application.
- System modelling with NuSMV probably not useful here - the design is okay.

Exercise 2: Specification (12 points)

Write formal specifications for the following informal requirements in an appropriate specification formalism of your choice (4 points per item). You may assume that appropriate atomic propositions, query methods and classes exist.

a The dog remains black

b Baby eyes may change colour

c Baby eyes can change from blue to brown, but not from brown to blue.

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 black or `//@ invariant dogColour == BLACK;`

b EF colorChange

c
```
/*@ constraint \old(eyeColour) == BLUE ==>
                      eyeColour == BLUE ||
                      eyeColour = BROWN;
    constraint \old(eyeColour) == BROWN ==> eyeColour = BROWN;
  */
```

Exercise 3: Queueing Model (18 points)

Consider a waiting room with a capacity of N persons and two serving stations. The people in the room are waiting for one of the two stations to be available. Each station will signal that it is ready to serve the next person. The supervisor of the waiting room will send a client to the station and when the task is done, the clerk at the station will clean up and ask for the next person.

A model of the station is given below:

```
1  MODULE Station(new)
2    DEFINE ready := state=idle;
3
4    VAR
5      state : {idle,working,cleaning};
6      done  : boolean;
7
8    ASSIGN
9      init(state):=idle;
10     init(done):=FALSE;
11     next(done):=case
12       state=working & next(state)=cleaning : TRUE;
13       TRUE:FALSE;
14     esac;
15     next(state):=case
16       state=idle&new : working;
17       state=working : {working,cleaning};
18       state=cleaning : {cleaning,idle};
19       TRUE:state;
20     esac;
```

a (*8 pnts.*) Write a SMV module for the WaitingRoom.
The WaitingRoom module has three inputs:

new boolean that indicates that a new client has shown up.

ready1 boolean that indicates that station 1 is ready to start a a new client.

ready2 boolean that indicates that station 2 is ready to start a a new client.

The WaitingRoom has four outputs (can be DEFINE or VAR):

full boolean that indicates that the waiting room is full.

size integer that holds the number of people in the room.

next1 boolean that indicates that the supervisor sends a client to station 1.

next2 boolean that indicates that the supervisor sends a client to station 2.

b (*4 pnts.*) Write a main module that

- instantiates one waiting room and two stations, and
- that feeds the waiting room with clients, if it is not full.

c (*3 pnts.*) Write a property that says that if no new customers show up then eventually the waiting room will be empty.

d (*3 pnts.*) Why will the liveness property in the previous question fail when using the given station module? How would you fix the problem?

Answers

a (8 points) The next in full is needed, but no points deducted if it is missed.

```
MODULE WaitingRoom(new,ready1,ready2)
  DEFINE N := 37;
  DEFINE full := (next(size)=N);

  VAR
    size  : 0..N;
    next1 : boolean;
    next2 : boolean;

  ASSIGN
    init(size) := 0;
    init(next1) := FALSE;
    init(next2) := FALSE;
    next(next1) := case
      ready1 & size=1 & ready2 : {TRUE,FALSE};
      ready1 & size > 0 : TRUE;
      TRUE : FALSE;
    esac;
    next(next2) := case
      ready2 & size=1 & next(next1) : FALSE;
      ready2 & size > 0 : TRUE;
      TRUE : FALSE;
    esac;
    next(size) := case
      size>1 & size<N : size+(new?1:0)-(next(next1)?1:0)-(next(next2)?1:0);
      size = N : size - (next(next1)?1:0) - (next(next2)?1:0);
      size = 0 : (new?1:0);
      size = 1 & (next(next1)|next(next2)) : (new?1:0);
      size = 1 : (new?2:1);
    esac;
```

b (4 points) The main module could be

```
MODULE main
  VAR
    new : boolean;
    room : WaitingRoom(new,station1.ready,station2.ready);
    station1 : Station(room.next1);
    station2 : Station(room.next2);
  ASSIGN
    next(new) := case
        room.full : FALSE;
        TRUE : {TRUE,FALSE};
    esac;
```

c (3 points) A consise way of stating this is

```
1 LTLSPEC G F (new | room.size=0);
```

d (3 points) The station may be loop forever in both the working and cleaning states. The solution is to use a fairness clause:

```
1   FAIRNESS state=idle;
```

Exercise 4: Software Model Checking (12 points)

Consider the following program

```
1 void error();
2
3 int main() {
4   int x=0;
5   x=x+1;
6   x=x+1;
7   if(x>2){error();}
8 }
```

Suppose we want to prove that the `error()` method is never called.

a (2 points) How would we have to instrument the program to express this property?

b (6 points) What would be the Boolean program for the `main` method if we apply the abstraction `x > 2`?

c (4 points) The predicate abstraction `x > 2` is insufficient to prove this property. Explain why any counter example for this abstraction is spurious.

a `assert (0)` in the then-part, before the call to `error()`

b
```
bool b =false;
b = b ? true : *;
b = b ? true : *;
if(b){assert(0); error();}
```

c Any counter example must take the then-part. But the concrete execution never takes the then-part, because `x` is never greater than 2.

Exercise 5: Abstraction (12 points)

Consider interface `Wallet`.

```
1 public interface Wallet {
2
3     public void pay(int price);
4
5 }
```

It is intended to model a wallet, containing some amount of money. It has a method `pay` that only should be callable when there is enough money in the wallet.

The next page shows two classes implementing this interface:

- `WizardWallet`
- `SavingWallet`

The `WizardWallet` implements the wallet of Harry Potter, containing Galleons, Sickles and Knuts. The informal documentation explains the value of the different coin types. The class `WizardAux` can be used to compute the worth in Galleons, Sickles and Knuts of an amount given in Knuts.

The `SavingWallet` has a special saving pocket. Every time a payment is made, 10 % of the value of the payment is put into this saving pocket (provided there is enough money in the wallet). The amount of money in the wallet may become negative, provided there are enough savings in it to compensate, *i.e.*, `money + savings` has to be non-negative.

a (4 points) Write an abstract specification for `Wallet`, modelling the intended behaviour of the `pay` method.

b (4 points) Add appropriate specifications to the class `WizardWallet` that are sufficient to prove that its implementation of the `pay` method respects the specification inherited from `Wallet`. You do *not* have to add specifications for the class `WizardAux`.

c (4 points) Add appropriate specifications to the class `SavingWallet` that are sufficient to prove that its implementation of the `pay` method respects the specification inherited from `Wallet`.

```
1 class WizardWallet implements Wallet {
2
3     private int galleon;
4     private int sickle;
5     private int knut;
6
7     /* Seventeen silver Sickles to a Galleon and twenty-nine Knuts to a Sickle.
8        Therefore 1 Galleon = 17 Sickles = 493 Knuts. */
9
10    public void pay(int price) {
11        galleon = galleon - WizardAux.toGalleons(price);
12        sickle = sickle - WizardAux.toSickles(price);
13        knut = knut - WizardAux.toKnuts(price);
14    }
15 }
16
17 class WizardAux {
18
19    public static int toGalleons(int amount) {
20        return amount / 493;
21    }
22
23    public static int toSickles(int amount) {
24        return (amount % 493) / 17;
25    }
26
27    public static int toKnuts(int amount) {
28        return (amount % 493) % 17;
29    }
30 }
```

```
1 class SavingWallet implements Wallet {
2
3     private int money;
4     private int saving;
5
6     //@ invariant money + saving >= 0;
7
8     public void pay(int price) {
9         money = money - price;
10        int new_save = price/10;
11        if (money >= new_save) {
12            money = money - new_save;
13            saving = saving + new_save;
14        }
15    }
16 }
```

Answer

a (4 points)

```
1  public interface Wallet {
2
3      //@ invariant amount >= 0;
4      //@ public instance model int amount;
5
6      /*@ requires price > 0;
7          requires amount - price >= 0;
8          ensures amount == \old(amount) - price;
9      */
10     public void pay(int price);
11
12 }
```

b (4 points) `//@ represents amount == 493 * galleon + 17 * sickle + knut;`

c (4 points) `//@ represents amount = money + saving;`

Exercise 6: Run-time Checking (8 points)

Each subquestion is worth 4 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 Consider class `ArrayMethods`. What will happen when the JML run-time checker is used to validate this class?

```
1  public class ArrayMethods {
2
3      int[] a;
4
5      public ArrayMethods(int[] a) {
6          this.a = a;
7      }
8
9      //@ ensures (\forall int i; 0 <= i && i < a.length; \result <= a[i]);
10     public int min() {
11         int i = 0;
12         int res = a[i];
13         while (i < a.length) {
14             if (a[i] > res) {
15                 res = a[i];
16             }
17             i++;
18         }
19         return res;
20     }
21
22     //@ ensures (\forall int i; 0 <= i && i < a.length; \result >= a[i]);
23     public int max() {
24         int i = 0;
25         int res = a[i];
26         while (i < a.length) {
27             if (a[i] < res) {
28                 res = a[i];
29             }
30             i++;
31         }
32         return res;
33     }
34
35     public static void main(String [] args) {
36         int[] a = {1, 1, 1, 1, 1};
37         ArrayMethods obj = new ArrayMethods(a);
38         System.out.println(obj.min());
39         System.out.println(obj.max());
40     }
41 }
```

b Consider class `Clock`. What will happen when the JML run-time checker is used to validate this class?

```
1  public class Clock {
2
3      /*@ spec_public */ private int hours;
4      /*@ spec_public */ private int minutes;
5
6      //@ invariant 0 <= hours && hours < 24;
7      //@ invariant 0 <= minutes && minutes < 60;
8
9      //@ requires 0 <= h && h < 24;
10     //@ requires 0 <= m && m < 60;
11     public Clock(int h, int m) {
12         hours = h;
13         minutes = m;
14     }
15
16     public void transferMinutesToHours() {
17         hours = (hours + minutes/60) % 24;
18         minutes = minutes % 60;
19     }
20
21     //@ requires m >= 0;
22     public void addTime(int m) {
23         minutes = minutes + m;
24         transferMinutesToHours();
25     }
26
27     public static void main (String [] args) {
28         Clock c = new Clock (12, 55);
29         c.addTime(4);
30         c.addTime(6);
31     }
32 }
```

Answers

a Nothing will happen, becasue this particular execution (where all array elements are 1) will not violate the specification – even though the methods `min` and `max` clearly do not respect its specification.

b An invariant violation will be reported, because when the `addTime` method calls `transferMinutesToHours`, the invariant is violated during the second call of `addTime` (with argument 6).

Exercise 5: Static Checking (16 points)

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

a (4 points) Consider class `IceCream`. What will happen when ESC/Java is used to validate this class?

```
1  public class IceCream {
2
3      /*@ spec_public */ private int size;
4
5      //@ invariant size >= 0;
6      //@ constraint size > 0 ==> \old(size) > size;
7
8      //@ requires n >= 0;
9      public IceCream(int n) {
10         size = n;
11     }
12
13     //@ requires size >= 1;
14     public void lick() {
15         size = size - 1;
16     }
17
18     //@ requires size >= 2;
19     public void bigLick() {
20         size = size - 2;
21     }
22
23     public int measureSize() {
24         return size;
25     }
26 }
```

b (4 points) Consider class `Building`. What will happen when ESC/Java is used to validate this class?

```
1  public class Building {
2
3      /*@ spec_public */ private int nr_of_windows;
4      /*@ spec_public */ private int nr_of_doors;
5
6      //@ invariant nr_of_doors >= 1;
7      //@ invariant nr_of_windows >= 1;
8
9      /*@ requires doors >= 1;
10         requires windows >= 1;
11     */
12     public Building(int doors, int windows) {
13         nr_of_doors = doors;
14         nr_of_windows = windows;
15     }
16
17     /*@ assignable nr_of_windows, nr_of_doors;
18         ensures nr_of_doors == \old(nr_of_doors) + 1;
19     */
20     public void addExtension() {
21         nr_of_doors++;
22         addWindow();
23     }
24
25     /*@ assignable nr_of_windows, nr_of_doors;
26         ensures nr_of_windows == \old(nr_of_windows) + 1;
27     */
28     public void addWindow() {
29         nr_of_windows++;
30     }
31
32     public void addBalcony() {
33     }
34
35 }
```

c (6 points) Consider class `Tree` (on the next page). The postcondition of method `count` states that this method counts the number of occurrences of the variable `x` in tree `t`. The specification uses the following methods:

- `subtrees`, returning the set of all subtrees of `t` – including `t` itself – , and
- `contains`, implementing set membership.

Add loop invariants that are sufficient to prove correctness of this method. You should have loop invariants describing the variables `examined`, `res` and `queue`. (ESC/Java does not support the `\num_of` keyword, but you may freely use it.) If you do not know how to phrase an invariant formally, then indicate it informally.

d (2 points) Give a specification that shows that the loop will terminate.

```
1 import java.util.*;
2
3 public class Tree {
4
5     private Tree left;
6     private Tree right;
7     private int val;
8
9
10    /*@ requires t != null;
11        ensures \result ==
12                (\num_of Tree t0; t.subtrees().contains(t0); t.val == x);
13    */
14    public static int count(Tree t, int x) {
15        int res = 0;
16        LinkedList<Tree> examined = new LinkedList<Tree>();
17        LinkedList<Tree> queue = new LinkedList<Tree>();
18        queue.add(t);
19        while (! queue.isEmpty()) {
20            Tree current = queue.remove();
21            examined.add(current);
22            if (current.val == x) {
23                res++;
24            }
25            if (current.left != null) {
26                queue.add(current.left);
27            }
28            if (current.right != null) {
29                queue.add(current.right);
30            }
31        }
32        return res;
33    }
34
35 }
```

Answers

a (4 points) Constraint violation by method `measureSize` because it will not always decrease the size of the ice cream.

b (4 points) The postcondition of `addExtension` might not be established, because the specification of `addWindow` states that it might modifies `nr_of_doors` as well.

c (2 points per correct loop invariant)

- All elements in the queue are non-empty.
- `res` is equal to the number of elements in `examined` whose value is equal to `x`.
- The trees that are in `queue` are not in `examined`
- All elements in `queue` and `examined` are subtrees of `t`.

d (2 points) `decrease t.size() - examined.size();`

Exercise 8: Test Generation with JML (12 points)

The next page has a simple implementation for changing a PIN on a smart card.

a (6 points) Provide a JML specification for the method changePin that is going to produce enough meaningful tests to thoroughly test this method. This is a security sensitive operation, so every part of the state of the PIN counts.

b (3 points) Also, at least one important property about the state of the PIN should hold at all times and regardless of the different specification cases you may think of. Specify this property as an invariant, rather than including it in the method specification.

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

```
1 public final class PinCard {
2
3     public final static int MAX_TRIES = 3;
4
5     public final static int STATUS_OK = 0;
6     public final static int STATUS_ERROR_TRY_EXHAUSED = 1;
7     public final static int STATUS_ERROR_WRONG_PIN = 2;
8     public final static int STATUS_ERROR_SAME_PINS = 3;
9
10    protected int pin;
11    protected int tryCounter;
12    protected boolean access;
13
14    /** Changes the current pin to newPin
15      *   @param oldPin - pin code for user verification
16      *   @param newPin - new pin to be assigned to the card
17      */
18    public int changePin(int oldPin, int newPin){
19      int result;
20      this.access = false;
21
22      if(tryCounter == 0){
23        result  = STATUS_ERROR_TRY_EXHAUSED;
24      }else{
25        assert this.tryCounter > 0;
26        this.tryCounter--;
27        if(this.pin == oldPin){
28          this.tryCounter = MAX_TRIES;
29          this.access = true;
30          if(oldPin == newPin) {
31            result = STATUS_ERROR_SAME_PINS;
32          }else{
33            this.pin = newPin;
34            result = STATUS_OK;
35          }
36        }else{
37          result= STATUS_ERROR_WRONG_PIN;
38        }
39      }
40      return result;
41  }
42
43 }
```

```
1 public final class PinCard {
2
3     public final static int MAX_TRIES = 3;
4
5     public final static int STATUS_OK = 0;
```

```
6     public final static int STATUS_ERROR_TRY_EXHAUSED = 1;
7     public final static int STATUS_ERROR_WRONG_PIN = 2;
8     public final static int STATUS_ERROR_SAME_PINS = 3;
9
10    protected int pin;
11    protected int tryCounter;
12    protected boolean access;
13
14    /** Changes the current pin to newPin
15      *   @param oldPin - pin code for user verification
16      *   @param newPin - new pin to be assigned to the card
17      */
18
19    //@ invariant tryCounter >= 0 && tryCounter <= MAX_TRIES;
20
21    /*@ public normal_behavior
22          requires !this.tryCounter == 0;
23          ensures this.pin == \old(this.pin);
24          ensures !this.access;
25          ensures this.tryCounter == 0;
26          ensures \result == STATUS_ERROR_TRY_EXHAUSED;
27        also public normal_behavior
28          requires this.tryCounter > 0;
29          requires oldPin != this.pin;
30          ensures this.pin == \old(this.pin);
31          ensures !this.access;
32          ensures this.tryCounter == \old(this.tryCounter) - 1;
33          ensures \result == STATUS_ERROR_WRONG_PIN;
34        also public normal_behavior
35          requires tryCounter > 0;
36          requires oldPin == this.pin;
37          requires oldPin == newPin;
38          ensures this.pin == \old(this.pin);
39          ensures this.access;
40          ensures this.tryCounter == MAX_TRIES;
41          ensures \result == STATUS_ERROR_SAME_PINS;
42        also public normal_behavior
43          requires tryCounter > 0;
44          requires oldPin == this.pin;
45          requires oldPin != newPin;
46          ensures this.pin == newPin;
47          ensures this.access;
48          ensures this.tryCounter == MAX_TRIES;
49          ensures \result == STATUS_OK;
50      @*/
51    public int changePin(int oldPin, int newPin){
52      int result;
53      this.access = false;
54
```

```
55        if(tryCounter == 0){
56          result  = STATUS_ERROR_TRY_EXHAUSED;
57        }else{
58          assert this.tryCounter > 0;
59          this.tryCounter--;
60          if(this.pin == oldPin){
61            this.tryCounter = MAX_TRIES;
62            this.access = true;
63            if(oldPin == newPin) {
64              result = STATUS_ERROR_SAME_PINS;
65            }else{
66              this.pin = newPin;
67              result = STATUS_OK;
68            }
69          }else{
70            result= STATUS_ERROR_WRONG_PIN;
71          }
72        }
73        return result;
74    }
75
76 }
```