EXAM
System Validation
(192140122)
13:45 - 16:45
03-11-2014

- 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)

You are a consultant for a company that has problems with their software development. Clearly, more rigorous development and some use of formal methods would improve the quality of their software. However, during your first meeting, the boss has told you that he thinks all formal methods are completely useless. How would you go ahead to convince him otherwise.

Describe your approach in at most 200 words.

Answer

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

-

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) When all safes are occupied, you have to wait until somebody empties his or her safe.

b (3 points) A safe only can be opened if you know the correct code.

c (4 points) If the user of a safe changes, in between there has to be a moment when the safe is free.

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

b

c

Exercise 3: Warehouse Modelling (15 points)

Consider a warehouse with 2 kinds of items (A and B), and a maximum total capacity M. All items of type A and B have equal size.

a (*6 points*) Write an SMV model that models the warehouse system which accepts `sell` and `store` requests. Both requests have the amount and the type of items to be sold or stored as parameters. The behaviour of the system should account for the following:

- Items of any kind can be only sold if there is enough items of this kind stored in the warehouse. On successful sell the items are removed from the warehouse. Otherwise nothing happens.
- Items of any kind can be only stored if there is enough space in the warehouse. On successful store the items are added to the warehouse. Otherwise nothing happens.

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.

b (*3 points*) Specify that never more than M items are stored in the warehouse.

c (*3 points*) Specify that it is always possible to sell all items.

d (*3 points*) We wish to ensure that the model contains no runs in which the warehouse continuously tries to sell items or continuously tries to store items. Add some information to the model that ensures this.

Answer

```
MODULE controller(request,item,amount,M)

VAR
  a_count : 0..M;
  b_count : 0..M;

DEFINE
  total := a_count+b_count;

ASSIGN
  init(a_count):=0;
  init(b_count):=0;
  next(a_count):=case
    item=a & request=sell & a_count>= amount : (a_count - amount);
    item=a & request=store & total + amount < M : (a_count + amount);
    TRUE:a_count;
  esac;
  next(b_count):=case
    item=b & request=sell & b_count>= amount : (b_count - amount);
    item=b & request=store & total + amount < M : (b_count + amount);
    TRUE:b_count;
  esac;

MODULE main

DEFINE
  M := 7;

VAR
  ctrl:controller(next(request),next(item),next(amount),M);
  request:{store,sell};
  amount:1..M;
  item:{a,b};

DEFINE
  total := ctrl.total;

-- (B) never more than M items are stored in the warehouse.
LTLSPEC G (total <= M)

-- (C) it is always possible to sell all items.
CTLSPEC AG EF total = 0

-- (D) disregard sequences which contain only sells or only stores.
FAIRNESS request=store
FAIRNESS request=sell
```

Exercise 4: Software Model Checking (15 points)

Consider the "multiply by four" C program given below.

a (*6 points*) Write satabs annotations that reflect the following contract: Under the assumption the N is not negative, the function cannot return error (1) and upon termination y will contain $4 * \mathsf{N}$.

You do not need to copy the program to the answer sheet, if you write a line number followed by text then it is understood that the text has to be inserted at that line number.

b (*5 points*) Construct the boolean program when abstraction is using just the predicate

P0: $\mathsf{y} == 4 * \mathsf{N}$

c (*4 points*) Why will satabs fail to verify the annotated program successfully? What do you need to do to give satabs a chance to finish the verification?

```
1  int nondet_int(); // random result
2  int N=nondet_int();
3  int x=nondet_int();
4  int y=nondet_int();
5
6  int main(){
7
8    x=0;
9
10   y=0;
11
12   while(x<N){
13
14     x++;
15
16     y=y+4;
17
18   }
19
20   if (x > N) {
21
22     return 1; // ERROR
23   }
24
25   return 0; // SUCCESS
26 }
```

Answers

The version of the code below contains all specifications asked for, plus a loop invariant in line 13. With this loop invariant the code will verify successfully. Without it, it will reach maximum iterations without finding a solution.

```
1 int nondet_int(); // random result
2 int N=nondet_int();
3 int x=nondet_int();
4 int y=nondet_int();
5
6 int main(){
7   __CPROVER_assume(N>=0);
8   x=0;
9
10  y=0;
11
12  while(x<N){
13    assert(y==4*x);
14    x++;
15
16    y=y+4;
17
18  }
19
20  if (x > N) {
21     assert(0);
22     return 1; // ERROR
23  }
24  assert(y == 4*N);
25  return 0; // SUCCESS
26 }
```

The boolean program with just P0 is:

```
1 bool P0=*;
2 void main(){
3   P0=P0;
4   P0=*;
5   while(*){
6     P0=P0;
7     P0=P0?false:*;
8   }
9   if (*) {
10     P0=P0;return;
11  }
12  P0=P0;return;
13 }
```

Exercise 5: Abstraction (10 points)

Consider the classes `Car` and `Bicycle` and their specifications.

a (8 points) Write an interface `Vehicle` that abstracts these classes, i.e., both `Car` and `Bicycle` should be able to implement `Vehicle`. Add specifications to `Vehicle` that abstractly specify the common behaviour of `Car` and `Bicycle`.

b (2 points) Discuss how the specifications of `Car` and `Bicycle` have to be adapted.

```
1  class Car {
2
3      private int distance;
4
5      private int gasInTank;
6
7      /*@ requires time >= 0;
8          ensures functioning() ? distance == \old(distance) + speed * time :
9                                  distance == \old(distance);
10     */
11     public void drive(int time, int speed) {
12         // implementation
13     }
14
15     //@ ensures \result == (gasInTank >= 0);
16     public boolean functioning() {
17         return gasInTank >= 0;
18     }
19 }
```

```
1  class Bicycle {
2
3      private int distance;
4
5      private boolean flatTire;
6
7      /*@ requires time >= 0;
8          ensures functioning() ? distance == \old(distance) + speed * time :
9                                  distance == \old(distance);
10     */
11     public void drive(int time, int speed) {
12         // implementation
13     }
14
15     //@ ensures \result == !flatTire;
16     public boolean functioning() {
17         return !flatTire;
18     }
19 }
```

Answer

a
```
interface Vehicle {

    //@ model instance private int _distance;

    //@ model private boolean _functioning;

    /*@ requires time >= 0;
        ensures functioning() ? _distance == \old(_distance) + speed * time
                              _distance == \old(_distance);
    */
    public void drive(int time, int speed);

    //@ ensures \result == _functioning;
    public boolean functioning();
}
```

b Remove method specifications, add represents clauses.

```
class Car {

    private int distance;

    private int gasInTank;

    //@ represents _distance <- distance;
    //@ represents _functioning <- gasInTank >= 0;

    public void drive(int time, int speed) {
        // implementation
    }

    public boolean functioning() {
        return gasInTank >= 0;
    }
}
```

```
class AbsBicycle implements Vehicle {

    private int distance;
    private boolean flatTire;

    //@ represents _distance <- distance;
    //@ represents _functioning <- !flatTire

    public void drive(int time, int speed) {
        // implementation
    }

```

```
13      public boolean functioning() {
14          return !flatTire;
15      }
16  }
```

Exercise 6: Run-time Checking (15 points)

Consider a stub implementation of an `Iterator` class below (assume it has a correct implementation and that the `Collection` class is fully implemented and provides functionality that one would normally expect). One of the desired security properties of the iterator is that the program always checks for the availability of the element using the `hasNext` method before the actual element is taken from the collection with `next`. The method `hasNext` can be called several times without an intermediate call to `next`, but `next` cannot be called more than once without an intermediate call to `hasNext` and it cannot be called at all if there are no elements left in the collection.

a (5 points) Add ghost specifications to the `Iterator` class that ensure the above properties.

b (2*5 points) Explain what exactly happens when the two test methods `test1` and `test2` are executed with the Runtime Assertion Checker and the specifications you have provided.

```
1  public class Iterator {
2      public Iterator(Collection c) {
3        ... // initialise the iterator based on c
4      }
5
6      public boolean hasNext() {
7        boolean result = ...; // check if there are elements left in c
8        return result;
9      }
10
11     public Object next() {
12       return ...; // return the next element from c
13     }
14
15     public Object test1() {
16       Integer one = new Integer(1);
17       Iterator it = new Iterator(new Collection(
18           new Object[] {one, one, one, one}));
19       while(it.hasNext()) {
20         if(it.next() == one) { return it.next(); }
21       }
22       return null;
23     }
24
25     public Object test2() {
26       Collection c = new Collection();
27       for(int i = 1; i<=4; i++) c.add(new Integer(i));
28       Integer zero = new Integer(0);
29       while(it.hasNext()) {
30         if(it.next() == zero) { return it.next(); }
31       }
32       return null;
33     }
34 }
```

Answers

a
```
1  public class Iterator {
2      //@ public instance ghost asked;
3
4      //@ ensures !asked;
5      public Iterator(Collection c) {
6        ... // initialise the iterator based on c
7        //@ set asked = false;
8      }
9
10     //@ ensures asked <==> \result;
11     public boolean hasNext() {
12       boolean result = ...; // check if there are elements left in c
13       //@ set asked = result;
14       return result;
15     }
16
17     //@ requires asked;
18     //@ ensures !asked;
19     public Object next() {
20       //@ set asked = false;
21       return ...; // return the next element from c
22     }
23 }
```

b Execution of method `test2` will succeed without any RAC errors, because the `zero` element looked for is not in the collection and hence the two (forbidden) subsequent calls to `it.next()` will never be executed. Method `test1` will fail during the first iteration of the loop – the collection contains only the one element, which will be immediately found and the body of the `if` statement will call `it.next()` for the second time causing a failed `requires asked;` precondition on method `next`.

Exercise 7: Static Checking (15 points)

a (5 points) Consider the array filtering method in the code below. The method copies non-negative elements from array `A` to array `B` and negative elements to array `C`. It also calculates the amount of the elements copied to arrays `B` and `C` in fields `b` and `c`. Provide loop annotations (loop invariant and termination clause) for this loop. Make them as complete as possible. Also provide the post condition that specifies the final outcome of this method. (There is no need to copy the code in your answer sheet, give the specifications only.)

```
1  class FilterArray {
2
3    int[] A, B, C; // arrays
4    int b, c; // collected items in arrays B and C
5
6    void filterArray() {
7      b = 0; c = 0;
8      int a = 0;
9      while(a < A.length) {
10       if(A[a] >= 0) {
11         B[b] = A[a]; b++;
12       }else{
13         C[c] = A[a]; c++;
14       }
15       a++;
16     }
17   }
18
19 }
```

b (2∗5 points) (Explain your answers, without explanation no points will be awarded).

Consider the `Coordinate3D` class below. Find two problems that the Extended Static Checker will stumble on in this example:

```
1  public class Coordinate3D {
2    private /*@ spec_public @*/ int x, y, z;
3
4    //@ public invariant x * 3 + y * 5 == z;
5
6    //@ assignable \everything;
7    void recomputeZ() {
8       z = x * 3 + y * 5;
9    }
10
11   //@ assignable x, z;
12   void updateX(int n) {
13       x = x + n;
14       recomputeZ();
15   }
16 }
```

Answers

a The loop should be annotated in the following way:

```
1 class FilterArray {
2
3   int[] A, B, C; // arrays
4   int b, c; // collected items in arrays B and C
5
6   /*@ ensures b + c == A.length;
7       ensures (\forall int i; i>=0 && i<b; B[i] >= 0 &&
8             (\exists int j; j>=0 && j<A.length; B[i] == A[j]));
9       ensures (\forall int i; i>=0 && i<c; C[i] < 0 &&
10            (\exists int j; j>=0 && j<A.length; C[i] == A[j])); @*/
11  void filterArray() {
12    b = 0; c = 0;
13    int a = 0;
14    /*@ loop_invariant a >= 0 && a <= A.length;
15        loop_invariant a == b + c;
16        loop_invariant (\forall int i; i>=0 && i<b; B[i] >= 0 &&
17            (\exists int j; j>=0 && j<a; B[i] == A[j]));
18        loop_invariant (\forall int i; i>=0 && i<c; C[i] < 0 &&
19            (\exists int j; j>=0 && j<a; C[i] == A[j]));
20        decreases A.length - a; @*/
21    while(a < A.length) {
22      if(A[a] >= 0) {
23        B[b] = A[a]; b++;
24      }else{
25        C[c] = A[a]; c++;
26      }
27      a++;
28    }
29  }
30
31 }
```

b The first problem is that in method `updateX` the invariant is going to be violated when the method `recomputeZ` is called – its precondition that includes checking of the invariant is not satisfied when `x` is temporarily increased by `n`. The second problem is the assignable clause of `recomputeZ` that is too general and will not allow ESC to assume that `x` and `y` are not changed which is necessary to prove the `updateX` method correct.

Exercise 8: Test Generation with JML (10 points)

Below you find a simple implementation of a no-overdraw bank account with integer balance and a debit operation.

a (2 points) The class needs one invariant to be maintained for this kind of account, specify it.

b (5 points) Provide **one** complete specification case for the normal execution of the `debit` method, that is, when everything goes fine and the balance is debited without any exceptions being thrown. Specify preconditions and changes to the state suitable for generating a meaningful test case for this usage scenario.

c (3 points) For the complete testing, including all the exceptional cases, of the method `debit` there would be more specification cases needed. How many exactly for this method and why? Provide at least one set of test data for each of these specification cases.

```
1  public final class AccountBalance {
2
3      private int balance;
4
5      public void debit(int amount){
6        if(amount <= 0) throw new IllegalArgumentException();
7        if(this.balance - amount < 0) throw new NegativeBalanceException();
8        this.balance -= amount;
9      }
10
11 }
```

Answer

```
1  public final class AccountBalance {
2
3      private /*@ spec_public @*/ int balance;
4      //@ public invariant balance >= 0;
5
6      /*@ public normal_behavior
7            requires amount > 0;
8            requires this.balance - amount >= 0;
9            ensures this.balance == \old(this.balance) - amount;
10       @*/
11     public void debit(int amount){
12       if(amount <= 0) throw new IllegalArgumentException();
13       if(this.balance - amount < 0) throw new NegativeBalanceException();
14       this.balance -= amount;
15     }
16
17 }
```

There shall be 3 specification cases all together due to the three different execution branches that the code can take (two exceptional and one normal execution branch). The corresponding example test data would be:

this.balance	amount
10	−10
0	10
20	10