EXAM
Testing, Debugging, and Verification
TDA567/DIT082

DAY: 12 January 2015 TIME: $14^{00} - 18^{00}$

Responsible:	Moa Johansson (0702 455 015)
Results:	Will be published mid February
Extra aid:	Only dictionaries may be used. Other aids are *not* allowed!
Grade intervals:	**U**: 0 – 23p, **3**: 24 – 35p, **4**: 36 – 47p, **5**: 48 – 60p, **G**: 24 – 47p, **VG**: 48 – 60p, **Max.** 60p.

Please observe the following:

- This exam has 14 numbered pages.
  Please check immediately that your copy is complete
- Answers must be given in English
- Use page numbering on your pages
- Start every assignment on a fresh page
- Write clearly; unreadable = wrong!
- Fewer points are given for unnecessarily complicated solutions
- Read all parts of the assignment before starting to answer the first question.
- Indicate clearly when you make assumptions that are not given in the assignment

Good luck!

Assignment 1 (Testing) (15p)

(a) Briefly explain what *White Box* and *Black Box* testing is, and how they differ.

You work for a company that makes tiny robots used to investigate pipes, checking for damages. Your employer is considering using a new third party library for taking sensor readings and creating maps of pipes in sewer systems. The pipe is divided into chunks and is modelled by an array, with entries either 0, 1 or 2. If the array entry representing location i is 0, the robot has yet to examine it. If it is 1 the robot's sensor readings indicates the pipe is OK at location i. If it is 2 the robot's sensor readings have indicated that the pipe is damaged at location i.

(b) However, your boss is not convinced that this library has been tested to a sufficiently high standard and asks you to run some additional tests before deciding to adopt this new third party library. You do not have access to the library source code, only specifications of the methods. Describe the *methodology* you would use to *systematically* derive test-cases from the specification. Which inputs should you include and which could reasonably be excluded?

(c) Write down at least four actual test-cases for the method given below:

```
public int[] readSensor(int[] map, int sensorVal, int pos){...}
```

Its specification is as follows:
requires: `map` is a non-null array of length at least 1, with all entries either 0, 1 or 2. `sensorVal` is 1 or 2 and `pos` is some value between 0 and `map.length` - 1.
ensures: returns an array representing the map with the position `pos` updated to hold `sensorVal`, and all other values the same as before.

Give detailed explanations and motivations for your choices. You *must state how you defined and partitioned the input space*. Give your test-cases as triples of values:
`(map, sensorVal, pos) --> expected outcome.`

(d) Write down two test-cases for the small program below. Your test cases should satisfy *decision coverage* for the program.

```
if(x > y || x == 0)
   {res := 0;}
if (isEven(x))
   {res := x/2;}
return res;
```

(e) Does your test-suite from question (d) satisfy *condition coverage*? Motivate and explain why or why not.

Solution

[2p, 3p, 6p, 2p, 2p]

(a) Black box testing techniques use some *external description* of the software to derive test-cases, for example, a specification or knowledge about the input space of a method.

White box techniques derives tests from the *source code* of the software, for example from branching points in execution, conditional statements (e.g. if, while).

(b) Use a black-box technique, such as input space partitioning. For each method, first identify the *domain* of its inputs (this typically simply comes from the types of the inputs. Secondly, look at the preconditions of the method, to establish the *input space*. The preconditions may rule out some values of the full input domain, which are not allowed or for which the behaviour of the method is not specified (we can't know how it behaves on these inputs if its Java, or they are not allowed if its Dafny). Thus, there is no point deriving test-cases for these inputs. Finally, look at the postconditons of the method, to establish how it behaves for given inputs. This may suggest ways of *partitioning* the input space so that all inputs in the same partitioning cause the program to behave in a similar manner. Finally, choose values from each partitioning. Here it may be good to include border-cases.

(c) The *domains* for the inputs to the method are here:
`map --> all possible integer arrays`
`sensorVal, pos --> INTEGER.MIN_VALUE up to INTEGER.MAX_VALUE.`

From the preconditions of the specification, we can reduce the domain by define the input space for each of the three inputs:
`map --> non-null arrays with entries in {0,1,2} and length at least 1.`
`sensorVal --> {1,2} (Finite input space).`
`pos --> {0, 1,..., map.length-1} (Finite input space).`

Suggested partitionings:
`map --> arrays of length 1, length 2, length>2 with entries in {0,1,2}.`
`sensorValue:  1 and 2.`
`pos --> 0, 1, ..., map.length-1.`

From the above partitioning I choose the following four test-cases. Note that they should at least cover some different lengths of the array, and updates to the first, last and some middle element of the array. Some sensor reading should be 1 and some 2.

Test cases, e.g.
arrary of length 1, update first location (0) to 1.
`([0], 1, 0) --> [1]`

arrary of length 2, update last location (1) to 2.
`([2,1], 2, 1) --> [2,2]`

arrary of length > 2, update first location (0) to 2.
`([0,1,1], 2, 0) --> [2,1,1]`

array of length > 2, update middle location (2) to 1.
`([0,1,0,0], 1, 2) --> [0,1,1,0]`

(d) For decision coverage, each decision in the program needs at least one test case where it evaluates to true and one where it evaluates to false. E.g:
`{x --> 1, y --> 0}` (First decision is true, the second is false)
`{x --> 4, y --> 5}` (First decision is false, the second is true)

(e) For condition coverage, each condition in the program needs to have at least one test case where it evaluates to true and one where it evaluates to false. In our case, there are three conditions in the program. In the test-cases I've given above, condition coverage is not achieved as in both cases the condition `x == 0` evaluates to false. Full marks *only* if motivation is given showing that they know what condition coverage actually is. Simply answering yes/no is not sufficient.

Assignment 2 (Debugging) (10p)

The following Java method is intended to compute the minimum and maximum values occurring in an integer array.

```
1  static void minMax(int [] a) {
2    int min = 0;
3    int max = 0;
4    for(int i = 0; i < a.length; i++){
5      if(a[i] < min)
6        min = a[i];
7      if(a[i] > max)
8        max = a[i];
9    }
10   System.out.println(''Min: '' + min + ''\n Max: '' + max);
11 }
```

For an input array `[1,2,3]`, the program outputs:
`Min: 0`
`Max: 3`
Obviously, there is a defect in the program (maybe you've spotted it already?)

(a) When is a statement B *data dependent* on a statement A?

(b) When is a statement B *control dependent* on a statement A?

(c) When calling the method on the array `[1,2,3]`, which program statements is line 10 backward dependent on?

(d) Repair the method and correct any defects. Clearly state where changes have been made.

(e) Suppose you had been given a bug report, with a very large failing input, e.g. an array of length 10 000. **Briefly** describe how would you systematically go about finding a smaller failing test case, suitable for debugging?

Solution
[1p, 1p, 5p, 2p, 1p]
(a) B is data dependent on A iff (i) A writes to a location v that is read by B and (ii) there is at least one execution path between A and B in which v is not overwritten.

(b) B is control dependent on A iff B's execution is potentially controlled by A.

(c) `min` is backward data dependent on line 2 (we never reach line 6 on this input!), `max` is backward data dependent on lines 3 and 8.

`min` is backward control dependent on lines 4 and 5. `max` is backward control dependent on lines 4 and 7.

(d) The defect for the test-case `[1,2,3]` traces the defect back to the initialisation of `min` in line 2, which should not be `0`, but rather the first element of the array, `a[0]`. Note that although the method produce the correct answer for `max` for this test-case,

also the initialisation for `max` in line 3 is defective (consider a test-case `[-1,-2,-3]`). Thus, both lines 2 and 3 are infected. A corrected version of the method is:

```
1  static void minMax(int [] a) {
2    int min = a[0];
3    int max = a[0];
4    for(int i = 1; i < a.length; i++){
5      if(a[i] < min)
6        min = a[i];
7      if(a[i] > max)
8        max = a[i];
9    }
10   System.out.println(''Min: '' + min + ''\n Max: '' + max);
11 }
```

(e) To reduce the input, employ an algorithm for input minimisation, such as ddMin. It uses a divide-and-conquer strategy to split the input in smaller chunks recursively, until a small failing test case has been found.

Assignment 3 (Formal Specification) (13p)

For this question we consider a simple model of an airline ticket system written in Dafny. The model consists of two classes: `Ticket` and `Flight`:

```
class Ticket{
  var bookingRef : int;
  var checkedIn : bool;

  constructor (ref : int){}
}

class Flight{
  var seats : int;
  var nextBookingRef : int;
  var passports : array<int>;

  constructor(noSeats : int){}

  method IssueTicket(passportNo : int) returns (t : Ticket){}

  method CheckIn(passportNo : int, ticket : Ticket)
  modifies ticket;
  . . .
  {
    ticket.checkedIn := true;
  }
}
```

The `Ticket` class simply models a ticket, with a booking reference number and a boolean field stating if the traveller has checked in. The `Flight` class keeps track of how many seats there are on the flight (the `seat` field) and how many bookings has been made (the `nextBookingRef` field). Tickets issued for a given flight have `bookingRef`s in sequence, starting from 0, up to `seats-1`. The array `passports` maps the booking reference for each ticket issued so far to a corresponding passport number.

(a) Complete the body and specification of the constructor method for the `Ticket` class. A newly issued ticked should of course not yet be checked in.

(b) Complete the body and specification for the constructor method of the `Flight` class. The array `passports` should be initialised to 0 everywhere.
Hint: You might want to provide a predicate capturing the allowed values of the fields for `Flight` objects.

Continued on next page!

(c) Write down a specification and body for the `IssueTicket` method in the `Flight` class. The `Ticket` returned should have an appropriate `bookingRef` and the passport number should be appropriately connected to the ticket with that booking reference. Passport numbers have to be four digit positive numbers. Furthermore, at most one ticket can be purchased by each individual passenger.

(d) Provide a specification for the `CheckIn` method in the `Flight` class. The specification must be sufficiently strict so that a traveller is only allowed to check-in if providing a ticket and passport where the booking reference is valid for this flight, and the passport number provided at check-in matches the passport number associated with the ticket.

Solution

[1p, 3p, 4.5p, 4.5p]

```
class Ticket{
  var bookingRef : int;
  var checkedIn : bool;

  constructor (ref : int)
  modifies this;
  requires ref >= 0;
  ensures bookingRef == ref && !checkedIn;
  {
    bookingRef := ref;
    checkedIn := false;
  }
}

class Flight{

  var seats : int;
  var nextBookingRef : int;
  var passports : array<int>;

  predicate Valid()
  reads this;
  {
    passports != null &&
    passports.Length == seats &&
    0 <= nextBookingRef <= seats
  }
  constructor(noSeats : int)
  modifies this;
  requires noSeats > 0;
  ensures Valid();
  ensures seats == noSeats;
  ensures nextBookingRef == 0;
  ensures fresh(passports);
  ensures forall i :: 0 <= i < passports.Length ==> passports[i] == 0;
```

```
  {
    nextBookingRef := 0;
    seats := noSeats;
    passports := new int[seats];

    forall(i : int | 0 <= i < passports.Length){
      passports[i] := 0;
    }
  }

  method IssueTicket(passportNo : int) returns (t : Ticket)
  modifies `nextBookingRef, passports;
  requires Valid() && nextBookingRef < seats;
  requires 1000 <= passportNo <= 9999;
  requires forall i :: 0 <= i < nextBookingRef ==> passports[i] != passportNo;

  ensures fresh(t) && t.bookingRef == old(nextBookingRef)
          && !t.checkedIn;
  ensures nextBookingRef == old(nextBookingRef)+1;
  ensures Valid();
  {
    t := new Ticket(nextBookingRef);
    passports[nextBookingRef] := passportNo;
    nextBookingRef := nextBookingRef + 1;
  }

  method CheckIn(passportNo : int, ticket : Ticket)
  modifies ticket;
  requires Valid() && ticket != null;
  requires 0 <= ticket.bookingRef < seats;
  requires passports[ticket.bookingRef] == passportNo;
  ensures ticket.checkedIn;
  {
    ticket.checkedIn := true;
  }
}
```

Assignment 4 (Formal Verification) (12p)

The following small Dafny program computes the k^{th} positive even number, where the *first* even number is defined to be 0, the *second* is defined to be 2, the *third* is 4 and so on.

```
method kthEven(k : int) returns (e : int)
requires ?
ensures ?
 {
   e := 0;
   var i := 1;
   while (i < k)
   invariant ?
   {
     e := e + 2;
     i := i + 1;
   }
 }
```

(a) Complete the specification of the program by providing suitable `requires` and `ensures` clauses.

(b) State a suitable loop invariant for the program.

(c) State a suitable variant for the loop.

(d) Prove that the program satisfies the specification using the weakest precondition calculus.

Solution

[2p, 2p, 1p, 7p]

(a)

```
method kthEven(k : int) returns (e : int)
 requires k > 0;
 ensures e == 2 * (k-1)
```

(b) `invariant e == 2*(i-1) && i <= k`

(c) `decreases k-i`

(d) Full marks only given for clear derivations, stating which rules are applied, and providing justification for why each case holds. One mark for each correct step, two extra marks for clear detailed derivations everywhere.

Following the standard steps as in the lecture notes. We use the following abbreviations:

```
Pre: k > 0
Post: e == 2 * (k-1)
I: e == 2*(i-1) && i <= k
```

```
V: k-i
B: i < k
S1: e := 0;  i := 1;
S:  e := e + 2; i := i + 1;
```

1) Loop invariant holds on entry:

`Pre ==> wp(S1, I)`

Which expands to:

`k > 0 ==> wp(e := 0; i := 1, e == 2*(i-1) && i <= k)`

Apply the Seq-rule:

`k > 0 ==> wp(e := 0, wp( i := 1, e == 2*(i-1) && i <= k))`

Apply Assignment-rule:

`k > 0 ==> wp(e := 0, e == 2*(1-1) && 1 <= k)`

Apply Assignment rule:

`k > 0 ==> 0 == 2*0 && 1 <= k`

Which follows from `Pre`. As `k > 0`, it must be at least 1.

2) Loop invariant holds at each iteration

`I && B ==> wp(S, I)`

Which expands to:

```
e == 2*(i-1) && i <= k && i < k ==>
wp(e := e + 2; i := i + 1, e == 2*(i-1) && i <= k)
```

Apply the Seq-rule:

```
e == 2*(i-1) && i <= k && i < k ==>
wp(e := e + 2, wp( i := i + 1, e == 2*(i-1) && i <= k))
```

Apply Assignment:

```
e == 2*(i-1) && i <= k && i < k ==>
wp(e := e + 2, e == 2*(i+1-1) && i+1 <= k)
```

Apply Assignment:

```
e == 2*(i-1) && i <= k && i < k ==>
e+2 == 2*i && i+1 <= k
```

Let's consider each conjunct in the conclusion separately.

First, `e+2 == 2*i`. This follows from the corresponding conjunct in `I`, namely `e == 2*(i-1)` which expands to `e == 2*i - 2)`. Simply rearranging, we get `e + 2 == 2*i` as required.

Secondly, `i+1 <= k`. This is equivalent to saying that `i < k`. This in turn, follows directly from the loop guard, `i < k`.

3) Loop invariant holds on exit

After the loop exits, the program is finished, and the post-condition must hold.

```
I && !B ==> Post
```

Which expands to:

```
e == 2*(i-1) && i <= k && !(i < k) ==> e == 2 * (k-1)
```

From the conjunct `i <= k` in the invariant, and the negated loop guard `!(i < k)`, we can conclude that `i == k` when the loop exits (as `i` must be smaller or equal to `k`, but at the same time not smaller). Substituting `k` for `i` in the conclusion, it follows trivially from the invariant.

4) Loop variant bounded

```
I && B ==> V > 0
```

Which expands to:

```
e == 2*(i-1) && i <= k && i < k ==> k-i > 0
```

Which is equivalent to (simplifying the conclusion)

```
e == 2*(i-1) && i <= k && i < k ==> k > i
```

From the loop guard, we have that `i < k`. This implies to the conclusion, `k > i`.

5) Variant decreases

```
I && B ==> wp(V1 := V; S, V < V1)
```

Expand:

```
I && B ==> wp(V1 := k-i; e := e + 2; i := i + 1, k-i < V1)
```

Apply Seq-rule, followed by Assignment rule:

```
I && B ==> wp(V1 := (k-i); e := e + 2, k-(i+1) < V1)
```

Apply Seq-rule, assignment change nothing:

```
I && B ==> wp(V1 := (k-i), k-(i+1) < V1)
```

Apply Seq-rule and assignment again:

```
I && B ==> ==> k-i-1 < k-i
```

Which is trivially true.

Assignment 5 (Loop Invariant Generation) (10p)

Using recurrence solving, derive the polynomial equality invariants of the following loop:

```
i:=0; j:=a;
while (i < k) {
  i := i+2;
  j := j-4;
}
```

Solution

1) First, introduce a variable n for the loop counter. Variables i and j become functions: $i[n]$ and $j[n]$. The initial values give us $i[0] = 0$ and $j[0] = a$.

2) Set up a system of recurrences, from the updates in the loop body:
$i[n+1] = i[n] + 2$
$j[n+1] = j[n] - 4$

3) Compute the closed forms of the above:
$i[n] = i[0] + 2n$
$j[n] = j[0] - 4n$

4) Eliminate n and substitute in initial values:

From the equation for $i[n]$ we get that
$n = (i[n] - i[0])/2$, or, $n = (i - 0)/2 = i/2$

Substituting this value of n into the equation for $j[n]$ gives us:
$j = a - 2i$

This is our invariant.

(total 60p)