Re-Exam
Testing, Debugging, and Verification
TDA567/DIT082

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

Responsible:	Moa Johansson (0702 455 015)
Results:	Will be published in May
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 7 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) (12p)

(a) Name, and describe precisely, the coverage criteria, related to source code, that were described in the course.

(b) Describe in which way the various coverage criteria from (a) relate to each other. Explain your answers.

(c) Explain what White Box and Black Box testing is, and how they differ from one another.

Solution
[6p, 4p, 2p]

(a)

Statement Coverage (SC) is satisfied by a test suite TS, iff for every *node* n in the control flow graph there is at least one test in TS causing an execution path via n.

Branch Coverage (BC) is satisfied by a test suite TS, iff for every *edge* e in the control flow graph there is at least one test in TS causing an execution path via e.

Path Coverage (PC) is satisfied by a test suite TS, iff for every *execution path* ep of the control flow graph there is at least one test in TS causing ep.

Decision Coverage (DC) For a given *decision* d, DC is satisfied by a test suite TS if it contains at least two tests, one where d evaluates to *false*, and one where d evaluates to *true*. For a given *program* p, DC is satisfied by TS if it satisfies DC for all $d \in D(p)$.

Condition Coverage (CC) For a given *condition* c, CC is satisfied by a test suite TS if it contains at least two tests, one where c evaluates to *false*, and one where c evaluates to *true*. For a given *program* p, CC is satisfied by TS if it satisfies CC for all $c \in C(p)$.

Modified Condition Decision Coverage (MCDC) For a given *condition* c in decision d, MCDC is satisfied by a test suite TS if it contains at least two tests, one where c evaluates to *false*, one where c evaluates to *true*, d evaluates differently in both, and the other conditions in d evaluate identically in both. For a given *program* p, MCDC is satisfied by TS if it satisfies MCDC for all $c \in C(p)$.

(b)

- BC subsumes SC
- PC subsumes BC
- DC and CC are orthogonal
- DC subsumes BC
- MCDC subsumes DC

- MCDC subsumes CC

c)

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).

Assignment 2 (Debugging) (10p)

(a) Consider the following sequence of integer inputs: `0, 0, 0, 1, 1, 1, 0, 1`.
We want to use the `ddMin` algorithm to find a small failing input subsequence. An input sequence is failing if the number of `0`s in it is the same as the number of `1`s.
Give one complete `ddMin` derivation for the above input. Motivate each step, explaining what happens at each step as well as when and why the parameters of the algorithm changes. A derivation without accompanying explanations will not be given full marks.

(b) In the question (a), the `ddMin` algorithm was used to find a smaller failing input sequence. State three reasons for why finding a small failing input is relevant for debugging.

Solution
[7p, 3p]

(a) Full marks only with explanations of what goes on at each step.

Start with granularity $n = 2$ and sequence `[0, 0, 0, 1, 1, 1, 0, 1]`

```
==> n: 2 [1, 1, 0, 1] PASS (remove 1st chunk)
==> n: 2 [0, 0, 0, 1] PASS (remove 2nd chunk)
```

Increase number of chunks to *min(n*2, len([0, 0, 0, 1, 1, 1, 0, 1])) = 4* as all tests passed.

```
==> n: 4 [0, 1, 1, 1, 0, 1] PASS (remove 1st chunk)
==> n: 4 [0, 0, 1, 1, 0, 1] FAIL (remove 2nd chunk)
```

Adjust number of chunks to *max(n-1, 2) = 3* as one test failed.

```
==> n: 3 [1, 1, 0, 1] PASS (remove 1st chunk)
==> n: 3 [0, 0, 0, 1] PASS (remove 2nd chunk)
==> n: 3 [0, 0, 1, 1] FAIL (remove 3rd chunk)
```

Adjust number of chunks to *max(n-1, 2) = 2*

```
==> n: 2 [1, 1] PASS (remove 1st chunk)
==> n: 2 [0, 0] PASS (remove 2nd chunk)
```

Increase number of chunks to *min(n*2, len([0, 0, 1, 1])) = 4*

```
==> n: 4 [0, 1, 1] PASS
==> n: 4 [0, 1, 1] PASS
==> n: 4 [0, 0, 1] PASS
==> n: 4 [0, 0, 1] PASS
```

The number of chunks is now maximal, hence the minimal failing test case found here is:

```
==> [0, 0, 1, 1]
```

(b) The ddMin algorithm is used to compute a small failing test case (it is not always *the* smallest, as in the example above. This is because the algorithm is greedy). Having a small failing test-case is useful because:

- A small failing test-case makes debugging easier, there is less code exercised (or fewer loop iterations) so the space where the bug could possibly be is reduced.
- A small failing test-case might be easier to understand for a human.
- The failing test-case can be made part of a regression test suite, which is rerun periodically.

Assignment 3 (Formal Specification) (15p)

Consider an implementation of a circular buffer storing integers in Dafny. A circular buffer starts empty, with some predefined `capacity`, specifying how many elements it can hold. The `size` field stores information about how many elements are currently stored in the buffer. As the buffer is circular, it does not matter at which index in the buffer we start inserting elements, this is given by the `start` field. Elements can then be added one by one, as long as there is room in the buffer. Should we reach the end, we simply wrap around and start inserting elements from the beginning of the buffer, where there are still unallocated slots.

Below is a skeletal implementation of a class `CircularBuffer`:

```
class CircularBuffer{
  var buffer : array<int>;
  var capacity : int;
  var size : int;
  var start : int;

  predicate Valid()
  { }

  constructor(cap: int, startInd : int)
  { }

  method Add(elem : int)
  {
    var i := start+size;
    buffer[i%capacity] := elem;
    size := size+1;
  }
}
```

Continued on next page!

(a) Your task is to add the specifications and implementations that are currently missing, for `Valid`, the constructor method and the `Add` method, taking the following into account:

- `size` is never negative, and always less than, or equal to, `capacity`.
- The value of `capacity` and `buffer.Length` should always be the same.
- The `buffer` field should never be null in a `CircularBuffer` object.
- There should be space for at least one element in the circular buffer.
- On initialisation, the `startInd` must be between 0 and `capacity`. The newly allocated buffer contains only zeroes.
- As long as there is room in the buffer, i.e. `size` is strictly smaller than `capacity`, the following must hold:
  - `Add` increases the `size` by one.
  - After calling `Add(elem)`, `elem` is stored in the buffer at some valid index.

(b) Now suppose that we want to add a method `FindFirstOdd()`. This method should return the index of the first element in the buffer which is odd, counting from index 0 (i.e. you do not need to take the `start` index into account here). If there is no odd element in the buffer, it should return -1. Write down an implementation with pre-conditions, post-conditions and invariants to ensure it is correct.

```
method FindFirstOdd() returns (ind : int) {}
```

Solution

[]

```
class CircularBuffer{
 var buffer : array<int>;
 var capacity : int;
 var size : int;
 var start : int;

 predicate Valid()
 reads this, this.buffer;
 {
   buffer != null &&
   capacity > 0 && buffer.Length == capacity &&
   0 <= start < buffer.Length &&
   0 <= size <= capacity
 }
 constructor(cap: int, startInd : int)
 modifies this;
 requires cap > 0 && 0 <= startInd < cap;
```

```
  ensures fresh (buffer);
  ensures Valid();
  ensures start == startInd;
  ensures forall i :: 0 <= i < buffer.Length ==> buffer[i] == 0;
  {
    size := 0;
    capacity := cap;
    start := startInd;
    buffer := new int[cap];
    forall (i | 0 <= i < buffer.Length)
      { buffer[i] := 0;}

  }

  method Add(elem : int)
  modifies this, this.buffer;
  requires Valid();
  requires size < capacity;
  ensures Valid();
  ensures size == old(size)+1;
  ensures exists i :: 0 <= i < buffer.Length && buffer[i] == elem;
  {
    var i := start+size;
    buffer[i%capacity] := elem;
    size := size+1;
  }
}

 method FindFirstOdd() returns (ind : int)
  requires Valid();
  ensures 0 <= ind < buffer.Length ==> buffer[ind]%2 != 0 &&
          forall j :: 0 <= j < ind ==> buffer[j]%2 == 0;
  ensures ind == -1 ==>
         forall k :: 0 <= k < buffer.Length ==> buffer[k]%2 == 0;

  //One alternative solution
  requires Valid();
  ensures  -1 <= ind < buffer.Length;
  ensures 0 <= ind  ==> buffer[ind]%2 != 0;
  ensures forall j :: 0 <= j < ind ==> buffer[j]%2 == 0;
  ensures ind == -1 ==> forall k :: 0 <= k < buffer.Length ==> buffer[k]%2==0;
  {
    ind := -1;
    var i := 0;
    while (i < buffer.Length)
    invariant i <= buffer.Length;
    invariant forall j :: 0 <= j < i ==> buffer[j]%2 == 0;
    {
      if (buffer[i]%2 != 0)
      {
          ind := i;
          return;
      }
```

```
      i := i+1;
    }
  }
```

Assignment 4 (Formal Verification) (13p)

Consider the following Dafny program:

```
method AlwaysEven(x : int) returns (y : int)
ensures y%2 == 0;
{
 if (x%2 == 0)
  { y := x; }
else
  { y := (x-1);}
y := 2*y;
}
```

Next, suppose we want to run the following snippet of Dafny code:

```
method Test(){
  var m := AlwaysEven(2);
  var n := AlwaysEven(3);
  assert m == n;
}
```

(a) The above code will cause a Dafny compiler error:

```
Error:  assertion violation
```

Explain why.

(b) Fix `AlwaysEven` so that Dafny would be able to prove the assertion.

(c) Prove that your revised version of `AlwaysEven` satisfies its post-condition using the weakest pre-condition calculus. Show all details of your proof and motivate each step.

(d) Briefly explain what a *loop invariant* and a *loop variant* is, how they differ and what they are used for in verification of programs.

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

(a)

Dafny can't prove the assertion because outside of the body of `AlwaysEven` it only remember its contract. Therefore, inside the `Test` method, the only thing Dafny knows about `AlwaysEven` is that it always returns an even number. This restriction is an efficiency consideration to allow for the use of an automated theorem prover, otherwise, proofs would quickly become unfeasibly complicated.

(b)

You need to refine the contract by adding two extra post-conditions:

```
      ensures x % 2 == 0 ==> y == 2*x;
      ensures x % 2 == 1 ==> y == 2*(x-1);
```

(c)

```
Let R =
y%2 == 0 &&
x%2==0 ==> y == 2*x &&
x%2==1 ==> y == 2*(x-1)
```

Prove that:

```
wp(if(x%2==0) {y := x} else {y := x-1}; y:=2*y, R)

Apply Seq-rule:

wp(if(x%2==0) {y := x} else {y := x-1}, wp(y:=2*y, R))

Apply Assignment:
wp(if(x%2==0) {y := x} else {y := x-1}, R2)

where R2 =
2*y%2 == 0 &&
x%2==0 ==> 2*y == 2*x &&
x%2==1 ==> 2*y == 2*(x-1)

Apply Conditional-rule:

x%2==0 ==> wp(y:=x, R2) &&
x%2 != 0 ==> wp(y:=(x-1), R2)

Apply Assignment to the if-branch:

x%2==0 ==> 2*x%2 == 0 &&
      x%2==0 ==> 2*x == 2*x &&
                        x%2==1 ==> 2*x == 2*(x-1)

The first conjunct is trivially true. The second also.
The third is true because x%2==1 is false, and false ==> anything is
true.

Apply Assignment to the else-branch:
x%2 !=0 ==> 2*(x-1)%2 == 0 &&
       x%2==0 ==> 2*(x-1) == 2*x &&
                         x%2==1 ==> 2*(x-1) == 2*(x-1)
```

```
The first conjunct is trivially true.
For the second the premise x%2==0 is false, so the whole conjunct is
true. The third is trivial.
```

d) A loop invariant essentially encodes induction over the number of loop executions. It is a logical formula which has to hold immediately before the loop is entered, at each iteration of the loop, and immediately after the loop exits. It is used to ensure the correctness of the loop, with respect to the (method) specification. Proving this is referred to as *partial correctness*.

A loop variant is a statement which decreases at each iteration of the loop, and is bounded from below by 0. It is used to show that the loop terminates. If we also prove that the loop terminates, we have shown *total correctness*.