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 8 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.

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.

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) {}
```

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.