SAMPLE EXAM 1
Testing, Debugging, and Verification
TDA567/DIT082

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 17 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
- Indicate clearly when you make assumptions that are not given in the assignment

Good luck!

Assignment 1 (Testing) (12p)

(a) Consider the program below. It computes the length of the longest strictly increasing subsequence in the array.
Draw a *control-flow graph* for the method `longestIncrSeq` and use it to write down a test-suite which satisfies *branch coverage* for this program.
Write your test-cases in the format `array --> result`, where `array` is an integer array and `result` is the expected result on this input. Your test-suite should be *minimal* in the sense that no two inputs should cover the same branches.

```
// pre: arr is non-null and arr.length >= 1
// post: return the length of the longest uninterrupted
// increasing sequence in arr.

public static int longestIncrSeq(int[] arr) {
 if (arr.length == 1)
   return 1;
 else {
   int i = 1;
   int count = 1;
   int maxcount = 1;
   while (i < arr.length) {
        if (arr[i - 1] < arr[i])
          count = count + 1;
        else{
          if (count > maxcount)
             maxcount = count;
          count = 1;
        }
        i = i + 1;
  }
  if (count > maxcount)
    return count;
  else
    return maxcount;
  }
}
```

Continued on next page!

(b) Another coverage criteria is *decision coverage*. Briefly explain what this is, and indicate whether or not your test-suit from part (a) satisfies this criteria as well.

(c) In addition to decision coverage, we discussed another two kinds of logic coverage in class. Describe these. Also describe the relationship between the three logic based criteria.

Solution
[7p, 2p, 3p]

(a)
The control-flow graph is shown in Figure 1. 4 marks for the correctly drawn control graph, the reminder for a test suite which has branch coverage and is not overly complicated.

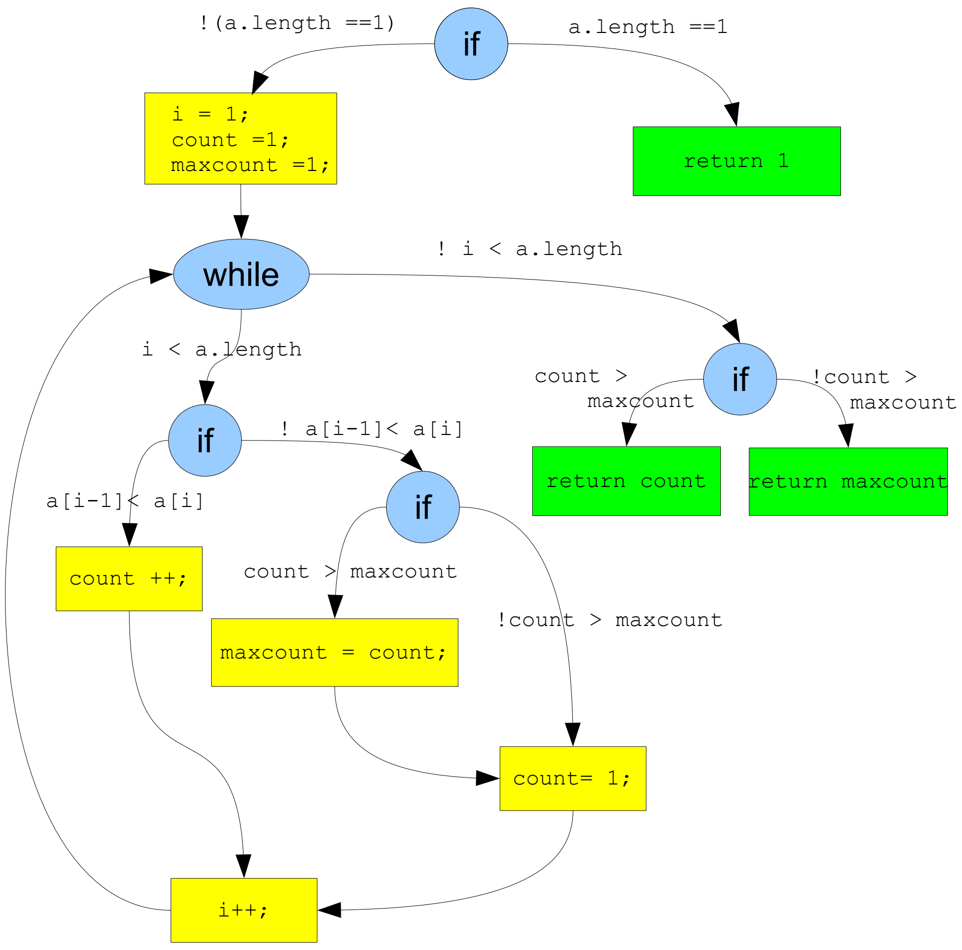


Figure 1: Control Flow

For Branch coverage, need 4 test cases, e.g.

```
[1] -> 1
[1,2] -> 2
[1,0] -> 1
[1,2,0] -> 2
```

(b)
For decision coverage, the test-suite must contain test-cases which cause each decision in the program (i.e. if-statements, loop guards) to evaluate to both true and false. As the test-suite in (a) satisfies branch-coverage, it will also have exercised all possible outcomes of decisions, so it satisfies decision coverage as well.

(c)
A correct answer should describe Condition Coverage and MCDC. For full marks, the student must also state the subsumption relationships between the different criteria:

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

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 conditions $c \in C(p)$.

- DC and CC are orthogonal (i.e. neither subsume the other)
- MCDC subsumes DC and CC.

Assignment 2 (Debugging) (12p)

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

(b) In the small Dafny program below, on which statement(s) is/are the statements in line 9 *data dependent*? Also state why.

```
1 method M(n : nat) returns (b : nat){
2       if(n == 0)
3          { return 0; }
4       var i := 1;
5       var a := 0;
6       b := 1;
7       while (i < n)
8       {
9        a, b := b, a+b;
11       i := i +1;
12      }
13 }
```

(c) On which statements is line 11 *backward dependent*? Also state why.

(d) The `ddMin` algorithm computes a minimal failure inducing input sequence. It relies on having a method `test(i)` which returns `PASS` if the input `i` passes the test or `FAIL` if the `i` causes failure (i.e. bug is exhibited).
Explain what we mean by *granularity* in the context of the `ddMin` algorithm.

(e) This question asks you to simulate a run of the `ddMin` algorithm. At each step, clearly state what the granularity is, how it was computed and why.
Suppose our input consists of sequences made out of the letters `A-Z`. Let `test` return `FAIL` whenever the sequence contains *two or more occurrences of the letter Z* somewhere in the sequence. The Z's does not need to be consecutive. Simulate a run of the `ddMin` algorithm and compute a minimal failing input from an initial failing input `[Z,B,R,Z,Z,Y,Z,X]`. Clearly state what happens at *each step* of the algorithm and what the final result is. Correct solutions without explanation will not be given the full score.

Solution
[1p, 2p, 2p, 1p, 6p]

(a)
B is control dependent on A, if B's execution is potentially controlled by A.

(b)
Line 9 is data-dependent on lines 5 and 6 as well as itself, as it may read the values of the previous iteration. 1 mark for lines 5 and 6, 2 marks if also line 9 is included in the answer.

(c)
Line 11 is backward dependent on lines 4 and 7 for the first iteration of the loop. On repeated iterations of the loop, it is backward dependent on lines 7 and on itself.

(d)
`ddMin` is a "divide and conquer" algorithm. The granularity decides in how many chunks we should divide the input into at each iteration. Initially, we start by splitting the input in two (granularity $n = 2$). If both halves pass the test, we must increase the granularity (number of chunks) to $min(n * 2, len(input))$, where n is the current granularity.

Similarly, we may have to decrease the granularity when we find a smaller chunk which fails the test to: $max(n - 1, 2)$, where n is the current granularity.

(e)
Full marks only if the answer motivates why the granularity changes as it does, and a motivation on the algorithms termination criteria. Start with granularity $n = 2$ and sequence `[Z,B,R,Z,Z,Y,Z,X]`. The algorithm will remove one chunk at the time. When it finds a failing test case, the algorithm will recurse on that input, i.e. it performs a depth-first search for the solution.

The number of chunks is 2
==> $\texttt{n} : 2, [\texttt{Z}, \texttt{Y}, \texttt{Z}, \texttt{X}]$ FAIL (take away first chunk)

After a failure, we adjust the number of chunks as follows:

Adjust number of chunks to $max(n - 1, 2) = 2$
==> $\texttt{n} : 2, [\texttt{Z}, \texttt{X}]$ PASS (take away first chunk)
==> $\texttt{n} : 2, [\texttt{Z}, \texttt{Y}]$ PASS (take away second chunk)

All tests passed, so we need to divide the input into smaller chunks. Increase number of chunks to $min(n * 2,\ len([\texttt{Z}, \texttt{B}, \texttt{R}, \texttt{Z}]) = 4$
==> $\texttt{n} : 4, [\texttt{Y}, \texttt{Z}, \texttt{X}]$ PASS (take away first chunk)
==> $\texttt{n} : 4, [\texttt{Z}, \texttt{Z}, \texttt{X}]$ FAIL (take away second chunk)

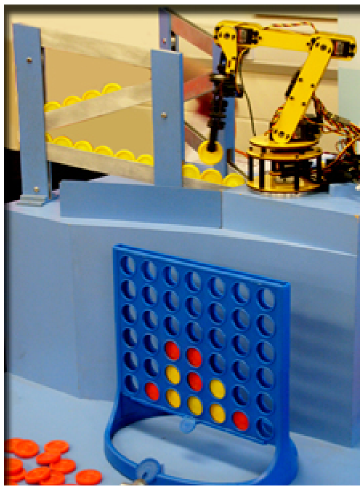
Adjust number of chunks to $max(n - 1, 2) = 3$
==> $\texttt{n} : 3, [\texttt{Z}, \texttt{X}]$ PASS (take away first chunk)
==> $\texttt{n} : 3, [\texttt{Z}, \texttt{X}]$ PASS (take away second chunk)
==> $\texttt{n} : 3, [\texttt{Z}, \texttt{Z}]$ FAIL (take away third chunk)

Adjust number of chunks to $max(n - 1, 2) = 2$
==> $\texttt{n} : 2, [\texttt{Z}]$ PASS (take away first chunk)
==> $\texttt{n} : 2, [\texttt{Z}]$ PASS (take away second chunk)

As $n == len([\texttt{Z}, \texttt{Z}])$ the algorithm now terminates with minimal failing input $[\texttt{Z}, \texttt{Z}]$

Assignment 3 (Formal Specification) (13p)

Connect Four is a two players game which takes place on a rectangular board placed vertically between them. One player has yellow tokens and the other red tokens. In each move, the player drops a token at the top of the board in one of the seven columns; the token falls down and fills the lowest unoccupied slot. Of course a player cannot drop a token in a column that is already full (i.e., it already contains six tokens). The goal is to connect four tokens vertically, horizontally, or diagonally.



We will represent the board by a two-dimensional array, called `board`. The different colours are represented by the integers 1 and 2. An empty slot is represented by the integer 0. Therefore, the above picture corresponds to the array depicted below.

5	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0
2	0	0	2	2	0	0	0
1	0	0	1	2	1	0	0
0	0	2	1	1	1	2	0
	0	1	2	3	4	5	6

The game is implemented in the Dafny class `Connect4`, see below. Note that the class is generic towards the dimensions of the board, and so should your specification be. To simplify your task of specification, we omit the general winning test, and instead consider a partial winning test, `WonHorizontally`. The methods `Drop` and `WonHorizontally` are *not* robust against wrong input. Consequently, the specifications must exclude wrong inputs, using `requires` clauses.

Continued on next page!

```
class Connect4{

  // Number of columns.
  var width : int;
  // Number of rows.
  var height : int;
  // The game board.
  var board : array2<int>;
  // The filling level of all columns.
  var level : array<int>;

  predicate Valid()
  reads this, this.level;
  {}

  method Init(w : int, h : int)
  modifies this;
  {
    width := w;
    height := h;
    board := new int[w,h];
    level := new int[w];

    //Initially empty board.
    forall(i,j | 0 <= i < width && 0 <= j < height){
      board[i,j] := 0;
    }
    forall(i | 0 <= i < width) {
        level[i] := 0;
    }
  }

  method Drop(player : int, column : int)
  modifies this.level, this.board;
  {}

  method WonHorizontally(player : int) returns (won : bool)
  {}
}
```

Continued on next page!

Your task is to enrich the `Connect4` class with Dafny specifications. You are **not** required to write implementations for `Drop` and`WonHorizontally`, only specifications.

Recall that you may access the dimensions of a two-dimensional array in Dafny using `Length0` and `Length1`.

(a) Complete the body for the predicate `Valid`. It should ensure `board` and `level` have been initialised, specify what data `board` and `level` can contain and finally also how the different fields depend on each other.

(b) Write a contract specifying the `Init` function. It should check that everything is initialised as expected and that legal values have been supplied for `w` and `h`. Furthermore, it should ensure that `board` and `level` are freshly allocated objects.

(c) Write a contract for `Drop`. It should capture the assumptions that `player` is either 1 or 2, and that the column is not yet full. The method places a token of the player at the lowest free slot in that column, leaving the rest of the board unchanged.

(d) Write a contract for `WonHorizontally`. It should capture the assumption that `player` is either 1 or 2 and return true if the player has won the game with four connected tokens in a *horizontal line*. I.e. if the player has won in some other way, this method still returns false.

Solution
[13p]

```
class Connect4{

  var width : int;
  var height : int;
  var board : array2<int>;
  var level : array<int>;

  predicate Valid()
  reads this, this.level;
  {
    board != null && level != null &&
    board.Length0 == width &&
    board.Length1 == height &&
    level.Length == width &&
    forall i :: 0 <= i < width ==> 0 <= level[i] < height &&
    forall i,j :: 0 <= i < width && 0 <= j < height ==> 0 <= board[i,j] <= 2
  }

  method Init(w : int, h : int)
  modifies this;
  requires w > 0 && h > 0;
  ensures fresh(board) && fresh(level);
  ensures width == w && height == h;
  ensures Valid();
  ensures forall i,j :: 0 <= i < width && 0 <= j < height ==> board[i,j] == 0;
  ensures forall i :: 0 <= i < width ==> level[i] == 0;
  {
    width := w;
    height := h;
    board := new int[w,h];
    level := new int[w];
    //Initially empty board.
    forall(i,j | 0 <= i < width && 0 <= j < height){
      board[i,j] := 0;
    }
    forall(i | 0 <= i < width) {
        level[i] := 0;
    }
  }

  method Drop(player : int, column : int)
  modifies this.level, this.board;
  requires this.Valid();
  requires 1 <= player <= 2;
  requires 0 <= column < level.Length;
  requires level[column] < height-1;

  ensures this.Valid();
  ensures level[column] == old(level[column]) + 1;
  ensures board[column,old(level[column])] == player;
  ensures forall i :: 0 <= i < level.Length && i != column ==>
              level[i] == old(level[i]);
  ensures forall i,j :: 0 <= i < board.Length0 && i != column &&
                        0 <= j < board.Length1 && j != level[column] ==>
              board[i,j] == old(board[i,j]);
  {. . .}

  method WonHorizontally(player : int) returns (won : bool)
  requires 1 <= player <= 2 && Valid();
  ensures won == exists i,j :: 0 <= i < width-3 && 0 <= j < height
  && board[i,j] == player && board[i+1,j] == player &&
     board[i+2,j] == player && board[i+3, j] == player;
  {. . .}
}
```

Assignment 4 (Verification and Test Generation) (12p)

```
method Max(arr : array<int>) returns (max : int)
requires ?
ensures ?
{
  // To be completed.
}
```

(a) Complete the above Dafny program which is supposed to compute the maximum of an array. In addition to the method body, your answer should state suitable pre- and post-conditions as well as loop invariants.

(b) **Briefly**, in concise English, explain what properties a loop invariant for *partial correctness* must satisfy.

(c) **Briefly**, in concise English, explain what *total correctness* is. What do we need, in addition to an invariant, to prove total correctness? State such an expression for the loop in the program above.

(d) When generating tests from specifications, it is normally required that the generated test inputs satisfy certain parts of the program specification. Which parts?

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

(a)

```
method Max(arr : array<int>) returns (max : int)
requires arr !=null && arr.Length > 0;
ensures forall i :: 0 <= i < arr.Length ==> max >= arr[i];
ensures exists i :: 0 <= i < arr.Length && max == arr[i];
{
  var i := 1;
  max := arr[0];
  while(i < arr.Length)
  invariant 0 < i <= arr.Length;
  invariant forall j :: 0 <= j < i ==> max >= arr[j];
  invariant exists j :: 0 <= j < i && max == arr[j];
  {
    if(arr[i] > max)
    {max := arr[i];}
    i := i +1;
  }
}
```

(b)
A loop invariant for partial correctness is a logical formula which hold in the state

before the loop is entered (implied by the precondition), holds at the beginning of each iteration of the loop, and holds after the exit of the loop (implies the postcondition).

(c)
For total correctness, we also need to prove that the program terminates. To prove this, we also need a *variant*, which is a formula which decrease (some measure) on each loop iteration. A variant for this program is `arr.Length - i`.

(d)
The preconditions, here `arr !=null && arr.Length > 0`

Assignment 5 (Verification) (11p)

This question concerns a program with a loop, which computes fibonacci numbers:

```
function fib(n : nat) : nat
{
  if (n==0) then 0 else
  if (n==1) then 1 else fib(n-1) + fib(n-2)
}

method Fib(x : nat) returns (res : nat)
ensures res == fib(x);
{
  if (x==0) { res := 0; }
  var i := 1;
  var nxt := 1;
  res := 1;
  while(i < x)
  invariant ?
  {
    res, nxt := nxt, nxt + res;
    i := i+1;
  }
}
```

(a) Give a loop *invariant* and a loop *variant* for the loop in the `Fib` method above. You may want to use the recursive function provided.

(b) Prove the `Fib` method correct using the weakest precondition calculus. You may assume that the Assignment rule works as expected for parallel assignments (i.e. as if we had written it using an intermediate variable).

Solution
[3p, 8p]

a)

```
function fib(n : nat) : nat
{
  if (n==0) then 0 else
  if (n==1) then 1 else fib(n-1) + fib(n-2)
}

method Fib(x : nat) returns (res : nat)
ensures res == fib(x);
```

```
{
  if (x==0) { res := 0; }
  var i := 1;
  var nxt := 1;
  res := 1;
  while(i < x)
  invariant 0 < i <= x;
  invariant res == fib(i);
  invariant nxt == fib(i+1);
  {
    res, nxt := nxt, nxt + res;
    i := i+1;
  }
}
```

Invariant: `res == fib(i) && nxt == fib(i+1) && 0< i <= x`
Variant: `x-i`

b)
The pre-condition is simply true. Let the R stand for the post-condition `res == fib(x)`. First apply the conditional rule: Let P stand for the code after the initial if-statement.

```
wp(if(x==0 {res := 0;} else P, res == fib(x))
```

by conditional rule

```
x==0 ==> wp(res :=0, res == fib(x))
&&
x != 0 ==> wp(P, res == fib(x))
```

The first conjunct is simple, we just apply the assignment rule to obtain:

```
x==0 ==> 0 == fib(x)
```

By the premise `x==0` and the definition of `fib` this reduce to `0==0` which is trivially true.

We then move on to prove the second conjunct: `x != 0 ==> wp(P, res == fib(x))` As this involves a loop, we follow the five steps from the lecture notes. We define the following abbreviations for clarity:

```
I : res == fib(i) && nxt == fib(i+1) && 0 < i <= x
V: x-i
R: res == fib(x)
S1: i := 1; nxt := 1; res := 1;
S: res, nxt := nxt, nxt + res; i := i+1;
```

1) Show that the invariant holds before entry of loop (recall that we still have the extra side condition that `x != 0` from the initial if-statement:

```
x != 0 ==> wp(S1, I)
```

Which expands to:

```
x != 0 ==>
   wp(i:=1; nxt:=1; res:=1, res==fib(i) && nxt==fib(i+1) && 0 < i <= x)
```

Apply seq rule, then assignment once for each variable. This instantiates the invariant expression to:

```
x != 0 ==> 1 == fib(1) && 1 == fib(1+1) && 0 < 1 <= x
```

As `x` is of type `nat` and not equal to `0`, it must be bigger than 0. Hence the last conjunct is true. By the def of `fib` the other two conjuncts are true:

```
x != 0 ==> 1 == 1 && 1 == 1 && 0 < 1 <= x
```

which is true.

2) Show that the loop invariant hold on each iteration:

```
x != 0 && I && B ==> wp(S, I)
```

Which expands to:

```
x != 0 && I && (i < x) ==> wp(res, nxt := nxt, nxt + res; i := i+1, I)
```

Apply the seq rule, then the assignment of `i` in to `i+1`:

```
x != 0 && res == fib(i) && nxt == fib(i+1) && 0 < i <= x && i<x ==>
   wp(res, nxt := nxt, nxt+res,
          res == fib(i+1) && nxt == fib((i+1)+1) && 0 < i <= x)
```

Apply assignment again, to the set `res --> nxt` and `nxt --> nxt+res`:

```
x != 0 && res == fib(i) && nxt == fib(i+1) && 0 < i <= x  && i<x ==>
          nxt == fib(i+1) && nxt+res == fib((i+1)+1) && 0 < i+1 <= x
```

Simplify using the equalities in the premises stating that `nxt==fib(i+1)` and `res==fib(i)`. The loop guard, `i < x`, simplifies the last conjunct,`i+1<=x` to true:

```
x != 0 && res == fib(i) && nxt == fib(i+1) && 0 < i <= x  && i<x ==>
          fib(i+1)==fib(i+1) && fib(i+1)+fib(i)==fib(i+2) && true
```

The only non-trivial conjunct is `fib(i+1)+fib(i)==fib(i+2)`. It follows from to the definition of fib.

3) Show that the post-condition is implied when the loop exits:

```
I && !B ==> R
```

Which expands to:

```
res == fib(i) && nxt == fib(i+1) && 0 < i <= x && !(i < x) ==>
   res == fib(x)
```

Among the premises we have the negated loop guard `!(i < x)` together with the conjunct `0 < i <= x` from the invariant, which imply that `i==x` when the loop exits. Hence R follow directly from the first conjunct of the premises:

```
res == fib(x) ==> res == fib(x)
```

4) Show that the variant is bounded from below by 0:

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

Which expands to:

`res == fib(i) && nxt == fib(i+1) && 0 < i <= x && i < x ==> x-i > 0`

As `i` is less than `x`, and `x` is positive (it is a natural number) then `x-1 > 0` is trivially true.

5) Show that the variant decrease at each loop iteration:

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

Apply seq rule.

`I && B ==> wp(V1 := x-i, wp(S, x-i < V1)`

Apply assignments in S, setting `i --> i+1`

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

Apply assignment

`I && B ==> x-i-1 < x-i)`

Trivially true.

(total 60p)