# EXAM
# Testing, Debugging, and Verification
# TDA567/DIT082

DAY:06 April 2018 TIME: 14:00 - 18:00

---

| | |
|---|---|
| Responsible: | Srinivas Pinisetty (Lecturer), Wolfgang Ahrendt (examiner) |
| 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,<br>**G**: 24 – 47p, **VG**: 48 – 60p, **Max.** 60p. |

**Please observe the following:**

- This exam has 15 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.
- **Rules of the weakest pre-condition calculus are provided in Page 15**.

# Good luck!

---

**Assignment 1 (Testing)** (16p)

(a) Consider the program below. It computes the length of the longest strictly increasing subsequence in the array. (7p)
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 suite from part (a) satisfies this criteria as well. (2p)

(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. (3p)

(d) Construct a minimal set of test cases for the code snippet below, which satisfies *Modified Condition Decision Coverage*. (4p)

```
int method1(int a, int b, int c)
{
        if ( (b > c && c == 6) || (a < 2) )
                return a;
        else
                return c;
}
```

**Solution**

[7p, 2p, 3p, 4p]

(a)

4 points for the correctly drawn control graph, and 3 points for a test suite which has branch coverage.

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 points, 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.

(d) {`a = 4, b = 1, c = 6`} {`a = 1, b = 1, c = 6`} {`a = 4, b = 7, c = 6`} {`a = 4, b = 7, c = 2`}

---

**Assignment 2 Debugging: Minimization using DDMin** (7p)

(a) 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). (2p)
Explain what we mean by *granularity* in the context of the `ddMin` algorithm.

(b) 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 G* somewhere in the sequence. The G's does not need to be consecutive. (5p)
Simulate a run of the `ddMin` algorithm and compute a minimal failing input from an initial failing input `[G,B,R,G,G,Y,G,X]`. Clearly state what happens at *each step* of the algorithm and what the final result is. Correct solutions without explanations will not be given the full score.

**Solution**
[2p, 5p]

(a)
`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.

(b)
Full points only if the answer motivates why the granularity changes as it does, and a motivation on the algorithm's termination criteria. Start with granularity $n = 2$ and sequence `[G,B,R,G,G,Y,G,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
==> `n` : 2, [`G`, `Y`, `G`, `X`] FAIL (take away first chunk)

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

Adjust number of chunks to $max(n - 1, 2) = 2$
==> `n` : 2, [`G`, `X`] PASS (take away first chunk)
==> `n` : 2, [`G`, `Y`] PASS (take away second chunk)

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

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

Adjust number of chunks to $max(n - 1, 2) = 2$
==> `n` : `2`, [`G`] PASS (take away first chunk)
==> `n` : `2`, [`G`] PASS (take away second chunk)

As $n == len$([`G`, `G`]) the algorithm now terminates with minimal failing input [`G`, `G`]

---

**Assignment 3 (Debugging: Backward dependencies)** (7p)

(a) When is a statement B *data dependent* on a statement A? (1p)

(b) When is a statement B *control dependent* on a statement A? (1p)

Consider the small Dafny program below:

```
1 method M1(n : nat) returns (b : nat){
2   if(n == 0)
3        { return 0; }
4   var a := 0;
5   var k := 1;
6   b := 1;
7   while (k < n)
8   {
9    a, b := b, a+b;
11   k := k+1;
12  }
13 }
```

(c) On which statement(s) is/are the statements in line 9 *data dependent*? On which statements is line 11 *backward dependent*? Also state why. (5p)

**Solution**

[1p, 1p, $2 + 3$p]

(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. More precisely, statement B is control dependent on statement A iff: (i) A is a control statement (while, for, if or else if), and (ii) Every path in the control flow graph from the start to B must go through A.

(c)

Line 9 is data-dependent on lines 4 and 6 as well as itself, as it may read the values of the previous iteration. 1 point for lines 4 and 6, 2 points if also line 9 is included in the answer.

Line 11 is backward dependent on lines 5 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.

---

**Assignment 4 (Formal Specification: Logic)** (4p)

(a) Consider the following propositional logic formula, where $p$ and $q$ are Boolean variables: (2p)

$$((p \vee q) \wedge (q \vee r)) \implies (p \vee r)$$

Is the above formula *satisfiable*? Is the above formula *valid*? Show and explain why?

(b) Represent each of the following English sentences in first-order logic, using reasonably named predicates, functions, and constants. (2p)

1. All entries in the array $a$ are greater than 0.
2. There is at least one occurrence of the number 7 in the array $a$.

**Solution**

(a)

| p | q | r | $p \vee q$ | $q \vee r$ | $p \vee r$ | $((p \vee q) \wedge (q \vee r))$ | Formula |
|---|---|---|---|---|---|---|---|
| T | T | T | T | T | T | T | T |
| T | T | F | T | T | T | T | T |
| T | F | T | T | T | T | T | T |
| T | F | F | T | F | T | F | T |
| F | T | T | T | T | T | T | T |
| F | T | F | T | T | F | T | F |
| F | F | T | F | T | T | F | T |
| F | F | F | F | F | F | F | T |

From the above truth table, we can notice that the formula $((p \vee q) \wedge (q \vee r)) \implies (p \vee r)$ can be true and is thus satisfiable, but it is not valid since it is not always true.

(b)

1. All entries in the array `a` are greater than 0.
$\forall\ i : int.\ 0 \leq i < a.Length \rightarrow a[i] > 0$

2. There is at least one prime number in the array `a`
$\exists\ i : int.\ 0 \leq i < a.Length \wedge a[i] = 7$

---

**Assignment 5 Formal Specification (1)** (6p)

Consider the following method that gives a *reversed* copy of an array in Dafny. For example, the result of running the method on an array containing `[2,4,3,1]` will be a new array containing `[1,3,4,2]`.

```
method reverse(a : array<int>) returns (res : array<int>)
        requires a != null
        ensures ?
        {
        var i := 0;
        res := new int[a.Length];
        while i < a.Length
                invariant ?
                {
                res[i] := a[a.Length - 1 - i];
                i := i + 1;
                }
        }
```

(a) Complete the specification of `reverse` by filling in the `ensures` field. (3p)

**Solution**

```
res != null && res.Length == a.Length && forall i : int :: 0 <= i < a.
 Length ==> res[i] == a[a.Length - 1 - i]
```

(b) Provide a loop invariant such that Dafny will be able to prove partial correctness. (3p)

**Solution**

```
invariant 0 <= i <= a.Length
invariant forall j :int :: 0 <= j < i ==> res[j] == a[a.Length - 1 - j]
```

---

**Assignment 6 Formal Specification (2)** (6p)

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

}
```

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

**Solution**
[6p]

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

---

**Assignment 7 (Formal Verification)** (14p)

This question is about verifying the following program:

```
y = x;
z = 0;
while (y > 0) {
        z = z + x;
        y = y - 1;
}
```

The program takes a integer $x$ as input and its specification is:

```
requires: x > 0
ensures: z = x * x
```

(a) **Briefly**, explain what properties a loop invariant for *partial correctness* must satisfy. (2p)

(b) Give a loop invariant for the while-loop in the program above (i.e. a loop invariant which suffices for proving partial correctness). (2p)

(c) Prove partial correctness of the above program using the weakest pre-condition calculus (the rules of the calculus are provided in the last page). (6p)

(d) To extend the verification from partial to *total correctness*, what needs to be proved in addition? (2p)

(e) Explain what a *variant* is, and state a variant for the while-loop in the above program. (2p)

**Solution**

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

(a) A loop invariant for partial correctness is a logical formula which

- Is implied by the precondition, hence it holds before the loop is entered.
- Is preserved by each iteration of the loop.
- Holds after the loop is exited and thus implies the post-condition.

(b) Loop invariant:

```
x * y + z = x * x & y >= 0
```

(c) **A. Initially Valid**

```
x > 0 -->
[y:=x, z:=0] Inv
```

Apply seq, and assignment rules and obtain the FOL formula:

```
x > 0 -> x * x + 0 = x * x & x > 0
```

To prove the two conjuncts:
`x * x + 0 = x * x`: follows by simplification and reflexivity.
`x > 0`: follows from the pre-condition.

**B. Invariant Preserved**
We have

```
{ Inv & y > 0 }
[]
z = z + x;
y = y - 1;
{ Inv }
```

Apply the assignment rule twice (followed by exit) to obtain:

```
x * y + z = x * x & y >= 0 & y > 0
-->
x * (y-1) + (z + x)  = x * x & (y-1) >= 0
```

First conjunct:
`x * (y-1) + (z + x) = x * x`
Expands to
`x * y - x + z + x = x * x`
simplifies to
`x * y + z = x * x`
which follows directly from the identical premiss.

Second conjunct:
The premise `y > 0` implies that `y-1 >= 0`.

**C. Use Invariant**

```
{Inv & ! y>0}
[]

{ z = x * x }
```

We obtain the FOL formula

```
x * y + z = x * x & y >= 0 & ! y>0
-->
z = x  * x
```

The premises `y >= 0` and `!y>0` implies that `y=0`. Setting `y=0` in the first conjunct in the premise simplifies it to `z = x * x`, from which the conclusion trivially follows.

(d) To prove total correctness we also need to prove that the loop terminates.

(e) A variant is an expression which decrease at each iteration of the loop, and that is bounded by some value (typically zero). A variant for our loop is simply `y`.

(total 60p)

## Additional Notes

Weakest pre-condition rules:

| | |
|---|---|
| Assignment: | $wp(x := e, R) = R[x \mapsto e]$ |
| Sequential: | $wp(S1; S2, R) = wp(S1, wp(S2, R))$ |
| Assertion: | $wp(assert\ B,\ R) = B\ \&\&\ R$ |
| If-statement: | $wp(\textit{if B then S1 else S2},\ R) =$ <br> $(B ==> wp(S1, R)) \wedge (!B ==> wp(S2, R))$ |
| If-statement (empty *else* branch): | $wp(\textit{if B then S1},\ R) =$ <br> $(B \rightarrow wp(S1, R))\&\&(!B ==> R)$ |
| While: | $wp(\textit{while B I D S},\ R) =$ <br> $I$ <br> $\wedge (B\ \&\&\ I ==> wp(S, I))$ <br> $\wedge (!B\ \&\&\ I ==> R)$ <br> $\wedge (I ==> D >= 0)$ <br> $\wedge (B\ \&\&\ I ==> wp(tmp := D; S, tmp > D))$ |