EXAM
Testing, Debugging, and Verification
TDA567/DIT083

DAY:12 January 2021 TIME: 08:30 - 12:30

Responsible:	Shaun Azzopardi (Examiner)
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:

- Answers must be given in English.
- 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 10**.

Good luck!

Assignment 1 (Testing) (16p)

Consider the program below. It computes the sum of numbers in an array up to a certain bound.

```
    // pre: bound > 0
    // post: return the sum of the floats in the array if it less than
the bound, or otherwise return the bound

    public static float boundedSum(float[] arr, int bound) {
        if (arr.length == 0){
            return 0;
        }
        else {
            int sum = 0;
            for (int i = 0; i < arr.length; i++) {
                sum += arr[i];
                if (sum > bound)
                    sum = bound;
                    i = arr.length;
                }
            }

            return sum;
        }
    }
```

Continued on next page!

When answering the questions below, write your test cases in the format `(arr, bound) --> result`, where `arr` is a float array, `bound` an integer, and `result` is the expected result on this input.

(a) Write down a test suite which satisfies *branch coverage* for this program. Your test suite should be *minimal* in the sense that no two inputs should cover the same branches. **Hint:** Drawing a control-graph of the program can help inspire your choice of test cases. (7p)

(b) Another coverage criterion is *decision coverage*. For `boundedSum`, motivate whether the test suite given in the previous question satisfies decision coverage. Explain if this is necessarily so. (3p)

(c) Yet another coverage criterion is *statement coverage*. For `boundedSum` give an example of a test suite that satisfies this criterion but that does not satisfy branch coverage. (2p)

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

```
int method1(int a, int b, int c)
{
    if ((c % 5 > 2 || b > a) && a > 20){
        return c;
    } else{
        return b;
    }
}
```

Assignment 2 Debugging: Minimization using DDMin (7p)

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 input `i` causes failure (i.e. bug is exhibited).

(a) Identify for these statements whether they are true or false and for each of them motivate your determination with an example and/or brief explanation: (2p)

1. ddMin starts with maximum granularity.
2. ddMin can only be applied to inputs that are explicitly of type array.
3. ddMin may not be able to reduce the input.

(b) Suppose our input consists of sequences of positive numbers (> 0). Let `test` return `FAIL` whenever the sequence contains *two or more* ***non-consecutive*** *occurrences of even numbers* somewhere in the sequence (i.e. [10,6] is non-violating, but [12,3,8] is violating). (5p)
Simulate a run of the `ddMin` algorithm and compute a minimal failing input from the initially failing input `[8,3,3,2,3,4,5,7]`. 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.

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

Consider the small Dafny program below:

```
1  method M2(n: nat, m: nat) returns(l: int){
2   if(m == 0){
3       return 0;
4   }
5   var l := 0;
6   if(n > m){
7       l := m;
8   }
9   var d := 0;
10  while(l < n){
11      l := l + 1;
12      d := (d + 1) % 2;
13  }
14  if(d % 2 != 1){
15      l := l*2;
16  }
```

(a) On which line/s is line 11 *data dependent*? Explain briefly why. (2p)

(b) On which line is line 11 *control dependent* on? Explain briefly why. (2p)

(c) On which statements is line 12 *backward dependent*? Also state why. (3p)

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

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

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

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. Arrays a and b only differ in at most 1 position.
2. Each element in array a is between 1 and 9.

Assignment 5 Formal Specification: Dafny (1) (6p)

Consider the following method that given a **non-empty integer array** returns a copy of the array with each element multiplied. For example, the result of running the method with a: `[2,4,3,1]` and b: `5` will be a new array containing `[10,20,15,5]`.

```
method multiply(a : array<int>, b: int) returns (res : array<int>)
requires ?
ensures ?
{
    var i := 0;
    res := new int[a.Length];
    while i < a.Length
    invariant ?
    {
        res[i] := b*a[i];
        i := i + 1;
    }
}
```

(a) Complete the formal specification of `reverse` by filling in the `requires` and `ensures` fields. (3p)

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

Assignment 6 Formal Specification: Dafny (2) (6p)

```
method firstLocationOf(a : array<int>, b : int) returns (l: int)
requires ?
ensures ?
{
    // To be completed.

}
```

Complete the above Dafny program, which is supposed to return the first location of a certain integer in an array (or otherwise return -1). In addition to the method body, your answer should include suitable pre- and post-conditions, as well as loop invariants.

Assignment 7 (Formal Verification) (14p)

This question is about verifying the following method:

```
method bound(x: int, y: int) returns(n: int)
ensures x >= y ==> (n == y)
ensures x <= y ==> (n == x)
{
    n := x;
    while(n > y){
        n := n - 1;
    }
}
```

(a) Give loop invariants for the while-loop in the program above (i.e. a loop invariant which suffices for proving partial correctness). **Hint: Avoid putting invariants in disjunctive form (i.e.** $A \vee B \vee C$**) to avoid a long proof.** (2p)

(b) Prove partial correctness of the above program using the weakest precondition calculus (the rules of the calculus are provided in the last page). (6p)

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

(d) Prove what remains to show total correctness **Hint #1:** You may need to increment by 1 any variant suggested by Dafny. **Hint #2:** You can attempt this even if you did not manage to show partial correctness. (4p)

(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:	$wp(\textit{assert } B, R) = B \;\&\&\; R$
If-statement:	$wp(\textit{if } B \textit{ then } S1 \textit{ else } S2, R) =$ $(B \texttt{==>} wp(S1, R)) \wedge (!B \texttt{==>} wp(S2, R))$
If-statement (empty *else* branch):	$wp(\textit{if } B \textit{ then } S1, R) =$ $(B \rightarrow wp(S1, R)) \&\& (!B \texttt{==>} R)$
While[1][2]:	$wp(\textit{while(B) invariant I; decreases V; } \{\ S\ \}, R) =$ I $\wedge\ (I \;\&\&\; B \texttt{==>} wp(S, I))$ $\wedge\ (I \;\&\&\; !B \texttt{==>} R)$ $\wedge\ (I \;\&\&\; B \texttt{==>} D > 0)$ $\wedge\ (B \;\&\&\; I \texttt{==>} wp(V_{old} := V; S, V_{old} > V))$

[1]Note that the rules for the while loop slightly differ from the rules available in the notes (page 9 and 15 in the Formal Verification 2 lecture slides), since in the notes we are considering the while in the context of a whole program, while the rules here deal with the while loop on its own. If W is the while loop, then showing $Q \implies wp(S1; W; S2, R)$ is equivalent to the rules in the notes.

[2]Note also that multiple invariant clauses can always be written as one invariant clause in conjunctive form.