Example EXAM
Testing, Debugging, and Verification
TDA567

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 12 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) **Briefly** describe the main features of the *Extreme Testing* methodology.

(b) Construct a minimal set of test-cases for the code snippet below, which satisfy *condition coverage*

```
if (a > b || b < 0)
   return a;
else
   return b;
```

(c) We discussed another two criteria for logic coverage in class. Describe these. Also describe the relationship between the three logic based criteria. Does your test-set from (b) satisfy any of the other coverage criteria. Why/why not?

Solution
[3p, 4p, 5p]

(a) Extreme testing is a development methodology where test-cases must be developed *before* the code is written. These tests must be re-run after every incremental code change.

(b)
Need two test-cases, for instance
{`a --> 2, b --> -1`} and {`a --> 2, b --> 3`}

(c)
This question checks if they have understood the logic based criteria. Full marks only if they give a satisfactory explanation to whether their answer to (b) satisfy any of the other criteria (mine above does not, as in both cases the decision comes out true). In addition, they should also give the following definitions of decision coverage and MCDC:

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 $d \in D(p)$.

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 $c \in C(p)$.

- 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 below small Dafny program, on which statement(s) is/are the statements in line 9 data dependent?

```
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*?

(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).
Suppose our input consists of representations of DNA sequences, made out out the letters `G,A,T,C` which represent the nucleotides. Let `test` return `FAIL` whenever the DNA sequence contains *two consecutive occurrences of the letter C* somewhere in the sequence.
Simulate a run of the `ddMin` algorithm and compute a minimal failing input from an initial failing input `[G,T,A,C,C,A,G,C]`. 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
[1*p*, 2*p*, 2p, 7p]

(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 reads the values of the previous iteration. 1 mark for lines 5 and 6, 2 marks if they also have line 9 in the answer.

(c)
Line 11 is backward dependent on lines 2,4 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)
Start with granularity $n = 2$ and sequence `[G,T,A,C,C,A,G,C]`.

The number of chunks is 2
==> `n` : 2, [`C`, `A`, `G`, `C`] PASS (take away first chunk)
==> `n` : 2, [`G`, `T`, `A`, `C`] PASS (take away second chunk)

Increase number of chunks to $min(n * 2,\ len([\texttt{G}, \texttt{T}, \texttt{A}, \texttt{C}, \texttt{C}, \texttt{A}, \texttt{G}, \texttt{C}])) = 4$
==> `n` : 4, [`A`, `C`, `C`, `A`, `G`, `C`] FAIL (take away first chunk)

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

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

Increase number of chunks to $min(n * 2,\ len([\texttt{A}, \texttt{C}, \texttt{C}, \texttt{A}]) = 4$
==> `n` : 4, [`C`, `C`, `A`] FAIL (take away first chunk)

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

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

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

Assignment 3 Formal Specification (1) (7p)

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.

Solution
[3p, 4p]

(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);
```

Assignment 4 (Formal Specification (2)) (15p)

For this question we consider a simple model of an airline ticket system written in Dafny. The model consists of two classes: `Ticket` and `Flight`:

```
class Ticket{
  var bookingRef : int;
  var checkedIn : bool;

  constructor (ref : int){}
}

class Flight{
  var seats : int;
  var nextBookingRef : int;
  var passports : array<int>;

  constructor(noSeats : int){}

  method IssueTicket(passportNo : int) returns (t : Ticket){}

  method CheckIn(passportNo : int, ticket : Ticket)
  modifies ticket;
  . . .
  {
    ticket.checkedIn := true;
  }
}
```

The `Ticket` class simply models a ticket, with a booking reference number and a boolean field stating if the traveller has checked in. The `Flight` class keeps track of how many seats there are on the flight (the `seat` field) and how many bookings has been made (the `nextBookingRef` field). Tickets issued for a given flight have `bookingRef`s in sequence, starting from 0, up to `seats-1`. The array `passports` maps the booking reference for each ticket issued so far to a corresponding passport number.

(a) Complete the body and specification of the constructor method for the `Ticket` class. A newly issued ticked should of course not yet be checked in.

(b) Complete the body and specification for the constructor method of the `Flight` class. The array `passports` should be initialised to 0 everywhere.
Hint: You might want to provide a predicate capturing the allowed values of the fields for `Flight` objects.

Continued on next page!

(c) Write down a specification and body for the `IssueTicket` method in the `Flight` class. The `Ticket` returned should have an appropriate `bookingRef` and the passport number should be appropriately connected to the ticket with that booking reference. Passport numbers have to be four digit positive numbers. Furthermore, at most one ticket can be purchased by each individual passenger.

(d) Provide a specification for the `CheckIn` method in the `Flight` class. The specification must be sufficiently strict so that a traveller is only allowed to check-in if providing a ticket and passport where the booking reference is valid for this flight, and the passport number provided at check-in matches the passport number associated with the ticket.

Solution
[2p, 3p, 5p, 5p]

```
class Ticket{
  var bookingRef : int;
  var checkedIn : bool;

  constructor (ref : int)
  modifies this;
  requires ref >= 0;
  ensures bookingRef == ref && !checkedIn;
  {
    bookingRef := ref;
    checkedIn := false;
  }
}

class Flight{

  var seats : int;
  var nextBookingRef : int;
  var passports : array<int>;

  predicate Valid()
  reads this;
  {
    passports != null &&
    passports.Length == seats &&
    0 <= nextBookingRef <= seats
  }
  constructor(noSeats : int)
  modifies this;
  requires noSeats > 0;
  ensures Valid();
  ensures seats == noSeats;
  ensures nextBookingRef == 0;
  ensures fresh(passports);
  ensures forall i :: 0 <= i < passports.Length ==> passports[i] == 0;
  {
    nextBookingRef := 0;
    seats := noSeats;
    passports := new int[seats];

    forall(i : int | 0 <= i < passports.Length){
      passports[i] := 0;
    }
  }

  method IssueTicket(passportNo : int) returns (t : Ticket)
  modifies ‘nextBookingRef, passports;
  requires Valid() && nextBookingRef < seats;
  requires 1000 <= passportNo <= 9999;
  requires forall i :: 0 <= i < nextBookingRef ==> passports[i] != passportNo;

  ensures fresh(t) && t.bookingRef == old(nextBookingRef)
          && !t.checkedIn;
  ensures nextBookingRef == old(nextBookingRef)+1;
  ensures Valid();
  {
    t := new Ticket(nextBookingRef);
    passports[nextBookingRef] := passportNo;
    nextBookingRef := nextBookingRef + 1;
  }

  method CheckIn(passportNo : int, ticket : Ticket)
  modifies ticket;
  requires Valid() && ticket != null;
  requires 0 <= ticket.bookingRef < seats;
  requires passports[ticket.bookingRef] == passportNo;
  ensures ticket.checkedIn;
  {
    ticket.checkedIn := true;
  }
}
```

Assignment 5 (Formal Verification) (14p)

The following small Dafny program computes the k^{th} positive even number, where the *first* even number is defined to be 0, the *second* is defined to be 2, the *third* is 4 and so on.

```
method kthEven(k : int) returns (e : int)
requires ?
ensures ?
 {
   e := 0;
   var i := 1;
   while (i < k)
   invariant ?
   {
     e := e + 2;
     i := i + 1;
   }
 }
```

(a) Complete the specification of the program by providing suitable `requires` and `ensures` clauses.

(b) State a suitable loop invariant for the program.

(c) State a suitable variant for the loop.

(d) Prove that the program satisfies the specification using the weakest precondition calculus.

Solution
[2p, 2p, 2p, 8p]

(a)

```
method kthEven(k : int) returns (e : int)
 requires k > 0;
 ensures e == 2 * (k-1)
```

(b) `invariant e == 2*(i-1) && i <= k`

(c) `decreases k-i`

(d) Full marks only given for clear derivations, stating which rules are applied, and providing justification for why each case holds. One mark for each correct step, two extra marks for clear detailed derivations everywhere.

Following the standard steps as in the lecture notes. We use the following abbreviations:

```
Pre: k > 0
Post: e == 2 * (k-1)
I: e == 2*(i-1) && i <= k
```

```
V: k-i
B: i < k
S1: e := 0;  i := 1;
S:  e := e + 2; i := i + 1;
```

1) Loop invariant holds on entry:

`Pre ==> wp(S1, I)`

Which expands to:

`k > 0 ==> wp(e := 0; i := 1, e == 2*(i-1) && i <= k)`

Apply the Seq-rule:

`k > 0 ==> wp(e := 0, wp( i := 1, e == 2*(i-1) && i <= k))`

Apply Assignment-rule:

`k > 0 ==> wp(e := 0, e == 2*(1-1) && 1 <= k)`

Apply Assignment rule:

`k > 0 ==> 0 == 2*0 && 1 <= k`

Which follows from `Pre`. As `k > 0`, it must be at least 1.

2) Loop invariant holds at each iteration

`I && B ==> wp(S, I)`

Which expands to:

```
e == 2*(i-1) && i <= k && i < k ==>
wp(e := e + 2; i := i + 1, e == 2*(i-1) && i <= k)
```

Apply the Seq-rule:

```
e == 2*(i-1) && i <= k && i < k ==>
wp(e := e + 2, wp( i := i + 1, e == 2*(i-1) && i <= k))
```

Apply Assignment:

```
e == 2*(i-1) && i <= k && i < k ==>
wp(e := e + 2, e == 2*(i+1-1) && i+1 <= k)
```

Apply Assignment:

```
e == 2*(i-1) && i <= k && i < k ==>
e+2 == 2*i && i+1 <= k
```

Let's consider each conjunct in the conclusion separately.

First, `e+2 == 2*i`. This follows from the corresponding conjunct in `I`, namely `e == 2*(i-1)` which expands to `e == 2*i - 2)`. Simply rearranging, we get `e + 2 == 2*i` as required.

Secondly, `i+1 <= k`. This is equivalent to saying that `i < k`. This in turn, follows directly from the loop guard, `i < k`.

3) Loop invariant holds on exit

After the loop exits, the program is finished, and the post-condition must hold.

```
I && !B ==> Post
```

Which expands to:

```
e == 2*(i-1) && i <= k && !(i < k) ==> e == 2 * (k-1)
```

From the conjunct `i <= k` in the invariant, and the negated loop guard `!(i < k)`, we can conclude that `i == k` when the loop exits (as `i` must be smaller or equal to `k`, but at the same time not smaller). Substituting `k` for `i` in the conclusion, it follows trivially from the invariant.

4) Loop variant bounded

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

Which expands to:

```
e == 2*(i-1) && i <= k && i < k ==> k-i > 0
```

Which is equivalent to (simplifying the conclusion)

```
e == 2*(i-1) && i <= k && i < k ==> k > i
```

From the loop guard, we have that `i < k`. This implies to the conclusion, `k > i`.

5) Variant decreases

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

Expand:

```
I && B ==> wp(V1 := k-i; e := e + 2; i := i + 1, k-i < V1)
```

Apply Seq-rule, followed by Assignment rule:

```
I && B ==> wp(V1 := (k-i); e := e + 2, k-(i+1) < V1)
```

Apply Seq-rule, assignment change nothing:

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

Apply Seq-rule and assignment again:

```
I && B ==> ==> k-i-1 < k-i
```

Which is trivially true.

(total 60p)