

EXAM
Testing, Debugging, and Verification
TDA567/DIT082

DAY: 09 January 2018

TIME: 08³⁰ – 12³⁰

Responsible: Srinivas Pinisetty (Lecturer), Wolfgang Ahrendt (examiner)

Contact: Srinivas Pinisetty (0733873221)
Exam room visit around 9:30 and 11:30

Results: Will be published mid February (or earlier)

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 21 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.
- **Weakest pre-condition rules are provided in Page 21.**

Good luck!

Assignment 1 (Testing)

(16p)

- (a) **Briefly** explain what *White Box* and *Black Box* testing is, and how they differ. (2p)
- (b) You work for a company that makes tiny robots. Your employer is considering using a new third party library for taking sensor readings. However, your boss is not convinced that this library has been tested to a sufficiently high standard and asks you to run some additional tests before deciding to adopt this new third party library. You don't have access to the library source code, only specifications of the methods. Describe the *methodology* you would use to *systematically* derive test-cases from the specification. (2p)
- (c) **Briefly** describe the main features of the *Extreme Testing* methodology. Also list **two** of its main advantages. (2p)
- (d) Write down **two** test-cases for the small program below. Your test cases should satisfy *decision coverage* for the program. (2p)

```
int method1(int x, int y)
{
    int res = 0;
    if((x == 0) || (x > y))
        res = y;
    if (isEven(x))
        res = x/2;
    return res;
}
```

- (e) For method1, does your test-suite from question (d) satisfy *condition coverage*? Motivate and explain why or why not. (2p)
- (f) We also discussed about *statement coverage*, and *branch coverage* criteria that are control-flow graph based. For method1, does your test suit from question (d) satisfy *statement coverage* and *branch coverage*. Motivate and explain why or why not. (2p)
- (g) Construct a minimal set of test-cases for the code snippet below, which satisfy *Modified Condition Decision Coverage*. (4p)

```
int method2(int a, int b, int c)
{
    if ( ( a < 3 ) || ( b > c && c == 5 ) )
        return a;
    else
        return c;
}
```

Solution

[2p, 2p, 2p, 2p, 2p, 2p, 3p]

(a) Black box testing techniques use some *external description* of the software to derive test-cases, for example, a specification or knowledge about the input space of a method. Black box testing techniques does not require the source code of the program to be tested.

White box techniques derives tests from the *source code* of the software, for example from branching points in execution, conditional statements (e.g. if, while).

(b) Use a black-box technique, such as input space partitioning. For each method, first identify the *domain* of its inputs (this simply comes from the types of the inputs). Secondly, look at the preconditions of the method, to establish the *input-space*. The preconditions may rule out some values of the full input domain, which are not allowed or for which the behaviour of the method is not specified (we can't know how it behaves on these inputs if its Java, or they are not allowed if its Dafny). Thus, there is no point deriving test-cases for these inputs. Finally, look at the postconditons of the method, to establish how it behaves for given inputs. This may suggest ways of *partitioning* the input space so that all inputs in the same partitioning cause the program to behave in a similar manner. Finally, choose values from each partitioning. Here it may be good to include border-cases.

(c) 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.

Advantages:

- Clear idea of what program should do before starting coding. Test-cases provide a specification for the program.
- May start with simple design and incrementally optimise later, without risking breaking the specification. Therefore may have rapid development process.

(d) For decision coverage, each decision in the program needs at least one test case where it evaluates to true and one where it evaluates to false. E.g:

{x --> 3, y --> 0} (First decision is true, the second is false)

{x --> 4, y --> 15} (First decision is false, the second is true)

(e) For condition coverage, each condition in the program needs to have at least one test case where it evaluates to true and one where it evaluates to false. In our case, there are three conditions in the program. In the test-cases I've given above, condition coverage is not achieved as in both cases the condition $x == 0$ evaluates to false.

NOTE: Simply answering yes/no is not sufficient. Full marks are given *only* if explanation/motivation is given showing that you understand what condition coverage actually is.

(f) Decision coverage implies branch coverage, and branch coverage subsumes statement coverage. Thus, the test suit from (d) satisfies both statement and branch coverage.

When we consider the control-flow graph of method1, for the test suit that I considered in (d), each node in the control-flow graph is visited in the execution path of at least one of the test cases. Similarly we will notice that each edge in the control-flow graph is visited in the execution path of at least one of the test cases.

(g) $\{a = 4, b = 1, c = 5\}$ $\{a = 1, b = 1, c = 5\}$ $\{a = 4, b = 7, c = 5\}$ $\{a = 4, b = 7, c = 2\}$

Assignment 2 Debugging: Minimization using DDMin (7p)

Consider a method that takes an array of integers as input, and computes a code that it returns as a result. The method *fails* if the input array consists of *two identical even numbers*. For example, the method fails when the input array is $[1, 2, 8, 6, 6, 2, 8, 5]$, $[2, 6, 7, 7, 5, 2]$.

- (a) A *1-minimal failing input* is an input where if you remove any single element, the resulting input succeeds. List *all* 1-minimal failing inputs in the following input array: $[1, 2, 8, 6, 6, 2, 8, 5]$. (2p)
- (b) Simulate a run of the `ddMin` algorithm and compute a 1-minimal failing input from the following initial failing input: $[1, 2, 8, 6, 6, 2, 8, 5]$. Clearly state what happens at *each step* of the algorithm and what the final result is. (5p)

Solution

(a) $[2, 2]$, $[6, 6]$, $[8, 8]$

(b) Start with granularity $n = 2$ and sequence $[1, 2, 8, 6, 6, 2, 8, 5]$.

The number of chunks is 2

==> $n : 2, [1, 2, 8, 6]$ PASS (take away first chunk)

==> $n : 2, [6, 2, 8, 5]$ PASS (take away second chunk)

Increase number of chunks to $\min(n * 2, \text{len}([1, 2, 8, 6, 6, 2, 8, 5])) = 4$

==> $n : 4, [8, 6, 6, 2, 8, 5]$ FAIL (take away first chunk)

Adjust number of chunks to $\max(n - 1, 2) = 3$

==> $n : 3, [6, 2, 8, 5]$ PASS (take away first chunk)

==> $n : 3, [8, 6, 8, 5]$ FAIL (take away second chunk)

Adjust number of chunks to $\max(n - 1, 2) = 2$

==> $n : 2, [8, 5]$ PASS (take away first chunk)

==> $n : 2, [8, 6]$ PASS (take away second chunk)

Increase number of chunks to $\min(n * 2, \text{len}([8, 6, 8, 5])) = 4$

==> $n : 4, [6, 8, 5]$ PASS (take away first chunk)

==> $n : 4, [8, 8, 5]$ FAIL (take away second chunk)

Adjust number of chunks to $\max(n - 1, 2) = 3$

==> $n : 3, [8, 5]$ PASS (take away first chunk)

==> $n : 3, [8, 5]$ PASS (take away second chunk)

$\implies n : 3, [8, 8]$ FAIL (take away third chunk)

Adjust number of chunks to $\max(n - 1, 2) = 2$

$\implies n : 2, [8]$ PASS (take away first chunk)

$\implies n : 2, [8]$ PASS (take away second chunk)

As $n == \text{len}([8, 8])$ the algorithm terminates with 1-minimal failing input $[8, 8]$

Assignment 3 (Debugging: Backward dependencies)

(7p)

The following Java method is intended to compute the minimum and maximum values occurring in an integer array.

```

1  static void minMax(int [] a) {
2    int min = 0;
3    int max = 0;
4    for(int i = 0; i < a.length; i++){
5      if(a[i] < min)
6        min = a[i];
7      if(a[i] > max)
8        max = a[i];
9    }
10   System.out.println(“Min: ” + min + “\n Max: ” + max);
11 }

```

For an input array [1,2,3], the program outputs:

Min: 0

Max: 3

Obviously, there is a defect in the program (maybe you’ve spotted it already?)

- (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)
- (c) When calling the method on the array [1,2,3], which program statements is line 10 backward dependent on? (3p)
- (d) Repair the method and correct any defects. Clearly state where changes have been made. (2p)

Solution

[1p, 1p, 3p, 2p]

(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) We can notice that `min` is backward data dependent on line 2 (we never reach line 6 with input array [1,2,3]!). `max` is backward data dependent on lines 3 and 8. `min` is backward control dependent on lines 4 and 5. `max` is backward control dependent on lines 4 and 7.

(d) The defect for the test-case [1,2,3] traces the defect back to the initialisation of `min` in line 2, which should not be 0, but rather the first element of the array, `a[0]`. Note that although the method produce the correct answer for `max` for this test-case,

also the initialisation for `max` in line 3 is defective (consider a test-case `[-1,-2,-3]`). Thus, both lines 2 and 3 are infected. A corrected version of the method is:

```
1  static void minMax(int [] a) {
2      int min = a[0];
3      int max = a[0];
4      for(int i = 1; i < a.length; i++){
5          if(a[i] < min)
6              min = a[i];
7          if(a[i] > max)
8              max = a[i];
9      }
10     System.out.println("Min: " + min + "\n Max: " + max);
11 }
```

Assignment 4 (Formal Specification: Logic) (6p)

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

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

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

- (b) Define the *pre* and *post* conditions for the following `linearSearch` method formally. Explain all predicates/functions that you use in your specification. (4p)

Informally, the `linearSearch` method should take a sorted array and search for the given number in the array. It should return -1 if the given number is not present in the array, and otherwise return an index such that the number is at that place in the array.

```
method linearSearch( a : array<int>, element : int)
    returns (index : int)
    {
        .....
        .....
    }
```

Solution

- (a)

p	q	$\neg p$	$p \wedge q$	$\neg p \vee q$	$(p \wedge q) \wedge (\neg p \vee q)$
T	T	F	T	T	T
T	F	F	F	F	F
F	T	T	F	T	F
F	F	T	F	T	F

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

- (b) From the informal specification, a pre-condition for the binary search method is that the input array should be sorted. Another implicit precondition is that the input array should not be null.

Pre-condition (requires):

$$a! = null \wedge \text{isSorted}(a)$$

where “isSorted” is a predicate that takes an integer array as input. It returns true if the input array is sorted, and false otherwise.

Regarding post-condition, when the method returns index that is different from -1 , then the element is at that index in the array. If the index returned is -1 , then the given element is not present in the array.

Post-condition (ensures):

$(a[\text{index}] = \text{element} \wedge \forall i : \text{int}, 0 \leq i < \text{index} \rightarrow a[i] \neq \text{element})$

\vee

$(\text{index} = -1 \wedge \forall i : \text{int}, 0 \leq i < a.\text{length} \rightarrow a[i] \neq \text{element})$

Assignment 5 Formal Specification

(9p)

For this assignment, consider a flight ticket booking system (we only consider a **simplified version** of a particular class). Someone has modeled a `FlightBooking` class as:

```

class FlightBooking{
    var name : string;
    var passportNum : string;
    var flightID: int;
    var ticketCode : int;

    predicate ticketCodeValid()
    requires name != null
    requires passportNum != null
    requires flightID != null
    {
        ticketCode == generateCode(name, passportNum, flightID)
    }

    constructor (na : string, pn : string, fl: int)
    requires na != null && pn != null
    ensures ticketCodeValid()
    {
        name := na;
        passportNum := pn;
        ticketCode := generateCode(na, pn, fl);
    }
}

function method generateCode(na: string, pn: string, fl: int)
    : int
requires na != null && pn != null && fl!= null
{ ... }

```

Bookings for a particular flight is modeled as a simple array of `FlightBooking`'s.

We now want a predicate that checks if an array of `FlightBooking`'s, that denote bookings for a particular flight journey is valid.

Bookings for a particular flight is a non-null array of `FlightBooking` where each element in the array is non-null, and the `flightID` corresponding to each element in the array is equal to the id of the flight, and and the ticket code corresponding to each booking is valid: the stored `ticketCode` is equal to the ticket code that is generated from the other details.

- (a) Write down the body of the (4p)
predicate `validateBookings(bookings :arr<FlightBooking>, flightID: int)`
`{ ... }`
Use Dafny syntax in your answer. We do not subtract points for minor syntactical errors.

Solution

```
bookings != null
&& forall i :: 0 <= i < bookings.length ==> bookings[i] != null
&& bookings[i].flightID == flightID
&& bookings[i].ticketCodeValid()
```

We now want to specify a method with the following type:

```
method checkin(bookingsFl :arr<FlightBooking>, passportNum :string,  
    tktCode: int, flightID: int) returns (checkinOK : bool)  
    requires ?  
    ensures ?
```

Informally, the `checkin` method takes a valid set of flight bookings corresponding to that flight, a non-null passport number, ticket code and flight id. It returns true if there is a booking corresponding to the given passport number, and the supplied ticket code is correct, and false in all other cases.

- (b) Write down the formal specification of `checkin`. In other words, fill in (5p) the `requires` and `ensures` clauses above. Use Dafny syntax in your answer. We do not subtract points for minor syntactical errors.

Solution

```
    requires passportNum != null && validateBookings(bookingsFl,  
flightID)  
    ensures checkinOK <==> exists i :: 0 <= i < bookingsFl.length  
&& bookingsFl[i].passportNum == passportNum  
&& bookingsFl[i].ticketCode == tktCode
```

Assignment 6 (Formal Verification)

(10p)

Consider the following Dafny program:

```

method AlwaysOdd(x : int) returns (y : int)
  ensures y%2 == 1;
{
    if (x%2 == 1)
    { y := x+1; }
    else
    { y := x+2;}
    y := (2*y)+1;
}

```

Next, suppose we want to run the following snippet of Dafny code:

```

method Test(){
    var m := AlwaysOdd(2);
    var n := AlwaysOdd(3);
    assert m == n;
}

```

- (a) The above code will cause a Dafny compiler error: (2p)

Error: assertion violation

Explain why.

- (b) Fix `AlwaysOdd` so that Dafny would be able to prove the assertion. (3p)
- (c) Prove that your revised version of `AlwaysOdd` satisfies its post-condition using the weakest pre-condition calculus. Show all details of your proof and motivate each step. (5p)

Weakest pre-condition rules are provided in Page 21.

Solution

[2p, 3p, 5p]

- (a)

Dafny can't prove the assertion because outside of the body of `AlwaysOdd` it only remember its contract. Therefore, inside the `Test` method, the only thing Dafny knows about `AlwaysOdd` is that it always returns an odd 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 == 1) ==> y == (2*(x+1))+1;

```

ensures $(x \% 2 == 0) \implies y == (2*(x+2))+1;$

(c)

Let $R =$
 $y \% 2 == 1 \ \&\&$
 $(x \% 2 == 1) \implies y == (2*(x+1))+1 \ \&\&$
 $(x \% 2 == 0) \implies y == (2*(x+2))+1$

Prove that:

$wp(\text{if}(x \% 2 == 1) \{y := x+1\} \text{ else } \{y := x+2\}; y := (2*y)+1, R)$

Apply Seq-rule:

$wp(\text{if}(x \% 2 == 1) \{y := x+1\} \text{ else } \{y := x+2\}, wp(y := (2*y)+1, R))$

Apply Assignment:

$wp(\text{if}(x \% 2 == 1) \{y := x+1\} \text{ else } \{y := x+2\}, wp(y := (2*y)+1, R2))$

where $R2 =$

$(2*y)+1 \% 2 == 1 \ \&\&$
 $(x \% 2 == 1) \implies (2*y)+1 == (2*(x+1))+1 \ \&\&$
 $(x \% 2 == 0) \implies (2*y)+1 == (2*(x+2))+1$

Apply Conditional-rule:

$x \% 2 == 1 \implies wp(y := x+1, R2) \ \&\&$
 $x \% 2 == 0 \implies wp(y := x+2, R2)$

Apply Assignment to the if-branch:

$((2*(x+1))+1) \% 2 == 1 \ \&\&$
 $((x \% 2 == 1) \implies (2*(x+1))+1 == (2*(x+1))+1)) \ \&\&$
 $(x \% 2 == 0) \implies (2*(x+1))+1 == (2*(x+2))+1$

The first conjunct is trivially true.

The second also true because the right-hans side of the implication is trivially true.

The third conjunct is true because $x \% 2 == 0$ is false, and $\text{false} \implies \text{anything}$ is true.

Apply Assignment to the else-branch:

$((2*(x+2))+1) \% 2 == 1 \ \&\&$


```
(x % 2 == 1) ==> (2*y(x+2))+1 == (2*(x+1))+1 &&  
(x % 2 == 0) ==> (2*(x+2))+1 == (2*(x+2))+1
```

The first conjunct is trivially true.

For the second the premise $x\%2==1$ is false, so the whole conjunct is true.

The third is true since the RHS of the implication is trivial.

Assignment 7 (Formal Verification (proving loops))

(5p)

Consider the following Dafny program:

```

method m1(n : nat) returns (i : nat)
requires n >= 0
ensures i == 2*n

{
    i := 0;
    while (i < n)
    invariant i <= n
    decreases n-i
    { i := i + 1; }
    i := 2*i;
}

```

Prove total correctness (including termination) for the above program using the weakest pre-condition calculus (Weakest pre-condition rules are provided in Page 21).

SolutionLet R be $i == 2*n$

To prove:

 $n \geq 0 \implies \text{wp}(i:=0; \text{while } (i < n) \dots \{ \dots \}; i := 2*i, i == 2*n)$ Let us first compute ‘‘ $\text{wp}(i:=0; \text{while } (i < n) \dots \{ \dots \}; i := 2*i, i == 2*n)$ ’’. $\text{wp}(i:=0; \text{while } (i < n) \dots \{ \dots \}; i := 2*i, i == 2*n)$

Apply sequential rule (thrice):

 $\text{wp}(i:=0, \text{wp}(\text{while } (i < n) \dots \{ \dots \}, \text{wp } (i := 2*i, i == 2*n)))$

Apply assignment rule:

 $\text{wp}(i:=0, \text{wp}(\text{while } (i < n) \dots \{ \dots \}, 2*i == 2*n))$

Simplify:

 $\text{wp}(i:=0, \text{wp}(\text{while } (i < n) \dots \{ \dots \}, i == n))$

Let us compute $\text{wp}(\text{while } (i < n) \dots \{ \dots \}, i = n)$.

We consider subgoals for computing weakest pre-condition of the loop.

Let us prove that the invariant is preserved $B \ \&\& \ I \implies \text{wp}(S, I)$. \blacksquare

$$(i < n \ \&\& \ i \leq n) \implies \text{wp}(i := i + 1, i \leq n)$$

Apply assignment rule:

$$(i < n \ \&\& \ i \leq n) \implies i + 1 \leq n$$

Simplify:

True

Prove that the failure of loop guard, and invariant implies the post-condition $\neg B \ \&\& \ I \implies i = n$.

$$(\neg(i < n) \ \&\& \ (i \leq n)) \implies i = n$$

Simplify:

True

Prove that the decrease expression D is bounded below by 0 $I \implies D \geq 0$. \blacksquare

$$(i \leq n) \implies (n - i \geq 0)$$

Simplify:

True

Prove that the decrease expression actually decreases

$$(B \ \&\& \ I) \implies \text{wp}(tmp := D; S, tmp > D)$$

$$((i < n) \ \&\& \ (i \leq n)) \implies \text{wp}(tmp := n - i; i := i + 1, tmp > n - i)$$

Apply sequential rule (twice)

$$((i < n) \ \&\& \ (i \leq n)) \implies \text{wp}(tmp := n - i, \text{wp}(i := i + 1, tmp > n - i))$$

Apply assignment rule:

$$((i < n) \ \&\& \ (i \leq n)) \implies \text{wp}(tmp := n - i, tmp > n - (i + 1))$$

Apply assignment rule:

$$((i < n) \ \&\& \ (i \leq n)) \implies (n - i) > n - (i + 1)$$

Simplify:

True

Thus we have

```
wp(i:=0; while (i<n)....{....}; i:= 2*i, i==2*n) =
  i <=n
  && (i < n && i<=n) ==> wp (i:=i+1, i<=n)
  && (!(i<n) && (i<=n)) ==> i==n
  && (i<= n) ==> (n-i >=0)
  && ((i<n) && (i<=n)) ==> wp(tmp :=n-i; i:=i+1, tmp > n-i)

= i <=n && True && True && True && True
```

Simplify
i<= n

We wanted to compute

```
wp(i:=0, wp(while (i<n)....{....}, i==n))
```

Substitute $wp(\text{while } (i<n)\dots\{\dots\}, i==n)$ with $i<= n$:
 $wp(i:=0, i<=n)$

Apply assignment rule:

```
0 <=n
```

We wanted to prove:

```
n >=0 ==> wp(i:=0; while (i<n)....{....}; i:= 2*i, i==2*n)
```

Substitute $wp(i:=0; \text{while } (i<n)\dots\{\dots\}; i:= 2*i, i==2*n)$ with $0 <=n$:
 $n >=0 ==> 0<=n$

True.

(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:	$wp(\text{assert } B, R) = B \ \&\& \ R$
If-statement:	$wp(\text{if } B \text{ then } S1 \text{ else } S2, R) =$ $(B \implies wp(S1, R)) \wedge (!B \implies wp(S2, R))$
If-statement (empty <i>else</i> branch):	$wp(\text{if } B \text{ then } S1, R) =$ $(B \rightarrow wp(S1, R)) \ \&\& \ (!B \implies R)$
While:	$wp(\text{while } B \text{ I } D \text{ S}, R) =$ I $\wedge (B \ \&\& \ I \implies wp(S, I))$ $\wedge (!B \ \&\& \ I \implies R)$ $\wedge (I \implies D >= 0)$ $\wedge (B \ \&\& \ I \implies wp(tmp := D; S, tmp > D))$