Testing, Debugging, and Verification exam
DIT082/TDA567

Day: 9 January 2016 Time: 14^{00} – 18^{00}

Responsible:	Atze van der Ploeg
Results:	Will be published mid February or earlier
Extra aid:	Only dictionaries may be used. Other aids are *not* allowed!
Grade intervals:	**U**: 0 – 18p, **3**: 19 – 24 p, **4**: 25 – 29p, **5**: 30 –37p, **G**: 19 – 29p, **VG**: 30 – 37p, **Max.** 37p.

Please observe the following:

- This exam has 14 numbered pages.
  Please check immediately that your copy is complete
- Answers must be given in English
- Please use page numbering on your pages
- Please write clearly
- Fewer points are given for unnecessarily complicated solutions
- Indicate clearly when you make assumptions that are not given in the assignment
- Answers to the exam will be published on the course website tomorrow.

Good luck!

1 Testing

Assignment 1 Certainty and Testing (2p)

In most cases, unit testing can give some assurances, but not guarantees for all inputs.

→ Explain why in most cases unit testing cannot give such hard guarantees.

Solution
Methods typically have an (practically) infinite number of inputs. It is hence impossible to have a unit test for each possible input.

Assignment 2 Coverage (4p)

Consider the following piece of Java code:

```
class Group{
  private final String[] names;

  Group(String[] names){
   this.names = names;
  }

  // requires: All elements of names are non-null
  // ensures: returns true if and only if
  //          there is an element in
  //          names that equals name
  boolean isPartOfGroup(String name){
    if(name == null) { return false; }
    for(int i = 0; i < names.length ; i ++){
      if(name.equals(names[i])){
        return true;
      }
    }
    return false;
}
```

→ Construct a Java class where the methods are tests of the `isPartOfGroup` method above, such that the test-cases together provide *statement coverage*.

Solution

```
class Test{
  static Group g = new Group(
    new String[]{"simon", "mauricio", "atze"});

  @Test void test1() {
   assertEquals(g.isPartOfGroup(null), false);
  }

  @Test void test2() {
   assertEquals(g.isPartOfGroup("mauricio"), true);
  }

  @Test void test3() {
   assertEquals(g.isPartOfGroup("Gustav"), false);
  }
}
```

Assignment 3 Mutation testing (4p)

Consider the following Java method:

```
/*
requires: input left and right are non-null arrays which are sorted
 in non-decreasing order
ensures: output is a non-null array, sorted in non-decreasing order,
 such that for any integer i, the number of occurrences in the output
 of i, is equal to the number of occurrences in the left arrays of i
 plus the number of occurrences in the right array of i.  */
public static int[] merge(int[] left, int[] right){
  int [] res = new int[left.length + right.length];
  int il = 0, ir = 0, i = 0;
  while(il < left.length && ir < right.length){
    if(left[il] <= right[ir]){
     res[i] = left[il];
     il += 1; i += 1;
    } else {
     res[i] = right[ir];
     ir += 1; i += 1;
    }
  }
  while (il < left.length) {
    res[i] = left[il];
    il += 1; i += 1;
  }
  while (ir < right.length) {
    res[i] = right[ir];
    ir += 1; i += 1;
  }
  return res;
}
```

Ludvig has constructed a set of tests for this method which consists of the following tests (in shorthand):

```
merge({},{}) == {}
merge({2,2,3},{1,1,1}) == {1,1,1,2,2,3}
merge({0,1,3,5},{2,4}) == {0,1,2,3,4,5}
```

Ludvig thinks that he does not need more tests: he cannot imagine a bug that he has not tested for. You, as a fresh expert on testing, do not agree with Ludvig.

→ Show that Ludvig is wrong: construct a mutant of the method that does not conform to the specification, but that is not killed by Ludvig's test set.

Solution

For example:

```
public static int[] merge(int[] left, int[] right){
  int [] res = new int[left.length + right.length];
  int il = 0, ir = 0, i = 0;
  while(il < left.length && ir < right.length){
    if(left[il] <= right[ir]){
     res[i] = left[il];
     il += 1; i += 1;
    } else {
     res[i] = right[ir];
     ir += 1; i += 1;
    }
  }
  while (il < left.length) {
    res[i] = left[il];
    il += 1; i += 1;
  }
  while (false) { // <- mutate here
    res[i] = right[ir];
    ir += 1; i += 1;
  }
  return res;
}
```

This code is wrong, but none of the tests from Ludvig's tests fail (kill the mutant).

Assignment 4 Test driven development (2p)

The test driven development methodology is often summarized as *red-green-refactor*.

→ Explain what *red-green-refactor* means.

Solution
The process is as follows:

1. Write tests, make sure they fail (red).
2. Implement method, make sure the tests succeed(green).
3. Clean up code (refactor).

Assignment 5 Minimization (5p)

Suppose we have method `f` which takes an array of characters as input and suppose that this method computes the output incorrectly if the input contains an even number of `'X'` characters (but not zero), and otherwise computes the result correctly.

The shortest example of a string which contains an even number of `'X'` characters is the string `"XX"`. However, Sven has used a correct implementation of the `ddMin` algorithm to minimize a failing example of the method `f`, and the result was not `"XX"` but `"XXXX"`.

(a) Explain why this is possible. (2p)

Solution
The `ddMin` algorithm computes a 1-minimal failing input, which means an input where if you remove any single character, the resulting input succeeds. To go from `"XXXX"` to `"X"` you need to remove *two* characters at the same time. So `ddMin` does not guarantee that it will reduce `"XXXX"` to `"XX"`.

(b) Simulate a run of the `ddMin` algorithm and compute a 1-minimal failing input from the following initial failing input: `[x,a,x,x,c,x,x,x]`. Clearly state what happens at *each step* of the algorithm and what the final result is. (3p)

Solution

Start with granularity $n = 2$ and sequence `[x,a,x,x,c,x,x,x]`.

The number of chunks is 2
==> $\texttt{n} : 2, [\texttt{x}, \texttt{a}, \texttt{x}, \texttt{x}]$ PASS (take away first chunk)
==> $\texttt{n} : 2, [\texttt{c}, \texttt{x}, \texttt{x}, \texttt{x}]$ PASS (take away second chunk)

Increase number of chunks to $min(n * 2,\ len([\texttt{x}, \texttt{a}, \texttt{x}, \texttt{x}, \texttt{c}, \texttt{x}, \texttt{x}, \texttt{x}])) = 4$
==> $\texttt{n} : 4, [\texttt{x}, \texttt{x}, \texttt{c}, \texttt{x}, \texttt{x}, \texttt{x}]$ PASS (take away first chunk)
==> $\texttt{n} : 4, [\texttt{x}, \texttt{a}, \texttt{c}, \texttt{x}, \texttt{x}, \texttt{x}]$ FAIL (take away second chunk)

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

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

Increase number of chunks to $min(n * 2,\ len([\texttt{l}, \texttt{f}, \texttt{o}, \texttt{o}]) = 4$
==> $\texttt{n} : 4, [\texttt{a}, \texttt{c}, \texttt{x}]$ PASS (take away first chunk)
==> $\texttt{n} : 4, [\texttt{x}, \texttt{c}, \texttt{x}]$ Fail (take away second chunk)

Adjust number of chunks to $max(n - 1, 2) = 3$
==> $\texttt{n} : 4, [\texttt{c}, \texttt{x}]$ PASS (take away first chunk)
==> $\texttt{n} : 3, [\texttt{x}, \texttt{x}]$ Fail (take away second chunk)

Adjust number of chunks to $max(n - 1, 2) = 2$
==> $\texttt{n} : 4, [\texttt{x}]$ PASS (take away first chunk)
==> $\texttt{n} : 3, [\texttt{x}]$ PASS (take away second chunk)

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

Assignment 6 Formal Specification (1) (3p)

CompCert is a *verified* compiler from C to assembly.

→ Briefly explain what we mean when we say that CompCert is a verified compiler from C to assembly. Use at least the following words in your answer: specification, behavior, proof.

Solution

CompCert has three ingredients:

- A formal specification of the C language, which states which input/output behaviors can be exhibited by a a program in C.
- A formal specification of the assembly language, which states which input/output behaviors can be exhibited by a a program in assembly.
- An executable mathematical function which translates a program in C to a program in assembly.

When we say that CompCert is verified, we mean that there is a proof that if a compiled version of a program (in assembly) can exhibit some behavior according to the specification of the assembly language, then this behavior can also be exhibited by the uncompiled version (in C) according to the specification of the C language. (Less accurate, but also OK is if a student says that there is a proof that that all input-output behaviors that can be exhibited by the source program can also exhibited by the target program)

Assignment 7 Formal Specification (2) (7p)

In this question you are going to specify and implement a method that gives a *reversed* copy of an array in Dafny. For example, the result of running the method on an array containing `[1,2,3,4]` will be a new array containing `[4,3,2,1]`. The header of the method is as follows:

```
method reverse(a : array<int>) returns (res : array<int>)
requires a != null
ensures ?
```

(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) Implement the `reverse` method. Use a `while` loop and provide a loop invariant and decrease clauses such that Dafny will be able to prove total correctness. (It is not allowed to use a parallel for loop.) (4p)

Solution

```
var i := 0;
res := new int[a.Length];
while i < a.Length
invariant 0 <= i <= a.Length
invariant forall j : int :: 0 <= j < i ==> res[j] == a[a.Length - 1 -
j]
{
  res[i] := a[a.Length - 1 - i];
  i := i + 1;
}
```

Assignment 8 (Formal Verification) (10p)

A remarkable fact of numbers is that the sum of the natural numbers 0 till n is $\frac{n(n+1)}{2}$. In other words (assuming $n \geq 2$):

$$0 + 1 + 2 + .. + n = \frac{n(n+1)}{2}$$

For example, $0 + 1 + 2 + 3 + 4 + 5 = \frac{5(5+1)}{2} = 15$

In this question, you are going to prove that $0 + .. + n = \frac{n(n+1)}{2}$ is true using the weakest-precondition calculus.

The expression $\frac{n(n+1)}{2}$ is implemented by `sumn`:

```
function sumn(n : int) : int { n * (n + 1) / 2}
```

The following method implements $0 + 1 + 2 + .. + n$:

```
method sum(n : nat) returns (s : nat)
ensures s == sumn(n)
{
    i := 0;
    s := 0;
    while i < n
    invariant i <= n && s == sumn(i)
    decreases n - i
    {
        i := i + 1;
        s := s + i;
    }
}
```

→ Prove total correctness (including termination) for the above program.

You can assume:

- $\texttt{sumn}(0) = 0$
- `s == sum(i) ==> s + (i + 1) == sum(i+1)`
  (or $\texttt{sumn}(i) + (i + 1) = \texttt{sumn}(i + 1)$)
  (below I explain why this is true in case you are interested, but this is not needed to make the exam.)

Recall that a method without an `requires` clause is the same as a method with the clause `requires true` .

Solution

Compute weakest postcondition :

`wp( i := 0; s := 0 ; while ..., s == sumn(n))`

Apply seq rule (x2)

`wp( i := 0, wp( s := 0 , wp(while ..., s == sumn(n))))`

Compute `wp(while ..., s == sumn(n))` first

`wp(while (i < n) (i <= n && s == sum(i)) (n-i) i := i + 1; s := s+i, s == sum(n))`

Which expands to (these should all hold):

1. Invariant holds before loop: `i <= n && s == sum(i)`
2. Invariant maintained in loop: `i < n && i <= n && s == sum(i) ==>`
   `wp(i := i + 1; s := s+i, i <= n && s == sum(i)`
3. Invariant and loop fail implies postcondition:
   `!(i < n) && i <= n && s == sum(i) ==> s == sumn(n)`
4. Decreases clause always positive : `i <= n && s == sum(i) ==> n - i >= 0`
5. Iteration decreases : `i < n && i <= n && s == sum(i) ==>`
   `wp(tmp := n-i; i := i + 1; s := s+i, tmp > n - i)`

Simplify (2):

```
i < n && i <= n && s == sum(i) ==>
wp(i := i + 1; s := s+i, i <= n && s == sum(i)
```

Compute `wp(i := i + 1; s := s+i, i <= n && s == sum(i))`

Apply seq rule:
`wp(i := i + 1, wp(s := s + i, i <= n && s == sum(i))`
Apply assignment rule (x2)
`i <= n && s+(i + 1) == sum(i+1)`

Plug in to (2):
`i < n && i <= n && s == sum(i) ==>`
`i <= n && s+(i + 1) == sum(i+1)`
Simplify using `i <= n` is both before and after ==> and remove `i < n`
`s == sum(i) ==> s+(i + 1) == sum(i+1)`
This is an assumption we had, so reduces to true.

Simplify (3) :

`!(i < n) && i <= n && s == sum(i) ==> s == sumn(n)`
Use `!(i < n) = i >= n`
`i >= n && i <= n && s == sum(i) ==> s == sumn(n)`
Use `i >= n \&\& i <= n <==> i == n`
`i == n && s == sum(i) ==> s == sumn(n)`
Use `i == n` in right hand side and remove `i == n`
`s == sum(i) ==> s == sumn(i)`
Simplify using `a ==> a == True`
`True`

Simplify (4)

`i <= n && s == sum(i) ==> n - i >= 0`
Remove irrelevant: `i <= n ==> n - i >= 0`
Rewrite `n - i >= 0 i <= n ==> n >= i`
Flip
`i <= n ==> i <= n`
Simplify using `a ==> a == True`
`True`

Simplify (5)

`i < n && i <= n && s == sum(i) ==>`
`wp(tmp := n-i; i := i + 1; s := s+i, tmp > n - i)`

Compute:

`wp(tmp := n-i; i := i + 1; s := s+i, tmp > n - i)`

Seq rule (x2)

`wp(tmp := n-i, wp(i := i + 1, wp( s := s+i, tmp > n - i)))`
Assignment rule (x 3)
`n - i > n - (i + 1)`
Simplify

`n - i > n - i - 1`
True by `a > a - 1 True`

Now 2,3,4,5 reduce to true. So the

```
wp(while ..., s == sumn(n))
= i <= n && s == sum(i)
```

Plug back into: `wp( i := 0, wp( s := 0 , wp(while ..., s == sumn(n))))` becomes:

`wp( i := 0, wp( s := 0 , i <= n && s == sum(i)))` Assignment (x2) `0 == sum(0)`
Use assumption `0 == 0` `True`

So weakest precondition of program is true. Now check that our precondition (true) implies that: `True ==> True` Which is `True`

This is the end of the exam, you do not need to read further to make the exam!

Below I explain why the assumptions above are true in case you are interested:

The assumption `s == sum(i) ==> s + (i + 1) == sum(i+1)`
follows from $\texttt{sumn}(i) + (i + 1) = \texttt{sumn}(i + 1)$
But why does this hold? Here is a proof:

$$\texttt{sumn}(n) + (n + 1) = \frac{n(n + 1)}{2} + (n + 1) = \frac{n(n + 1) + 2(n + 1)}{2}$$
$$\frac{(n + 2)(n + 1)}{2} = \texttt{sumn}(n + 1)$$