Testing, Debugging, and Verification re-exam
DIT082/TDA567

Day: 12 April 2017 Time: $14^{00} - 18^{00}$

Responsible:	Wolfgang Ahrendt Atze van der Ploeg Tel.: +316 81098446
Results:	Will be published mid May 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 12 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 Continuous integration (2p)

→ Briefly explain what continuous integration is.

Solution
Continuous integration means that a server periodically or on each commit checks out the code, builds the code and runs all the tests.

Assignment 2 Logic coverage (3p)

Consider the following piece of java code:

```
if (a > b && (x || c == 0) )
   return a;
else
   return b;
```

→ Construct a set of test-cases for the code snippet above, which satisfies *modified condition decision coverage (MCDC).*

Solution

```
{ {a = 0, b = 1, x = false, c = 1}, {a = 0, b = 1, x = true, c = 1},
{a = 0, b = 0, x = true, c = 1}, {a = 0, b = 1, x = false, c = 0}}
```

Assignment 3 Mutation testing (3p)

Consider the following Java method which counts the number of elements which are present in both input arrays:

```
/*

requires: input left and right are non-null arrays which are sorted
          in non-decreasing order
ensures: output is the number of elements that are present in
         both arrays
*/
public static int inBoth(int[] left, int[] right){
  int il = 0, ir = 0, res = 0;
  while(il < left.length && ir < right.length){
    if(left[il] == right[ir]) {
       il += 1; ir += 1; res += 1;
    } else if(left[il] < right[ir]) {
       il += 1;
    } else {
      ir += 1;
    }
  {
  return res;
}
```

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

```
inBoth({6,8,10},{}) == 0
inBoth({6,6,7},{4,5,6}) == 1
inBoth({3,4,5},{1,2,3,4}) == 2
inBoth({},{2,3,5}) == 0
```

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

This code is wrong, but none of the tests from Ludvig's tests fail (kill the mutant). None of the existing tests execute the second if clause (no statement coverage).

Assignment 4 Framing (2p)

In Dafny, it is required to state which variables are read (for functions) and which variables are modified (for methods).

→ Why does Dafny need this information?

Solution
This is needed for *efficiency* of (automatic) proofs. We know that a value of an expression only changes if a variable is modified that that expression reads.

Assignment 5 Logic and property based testing (4p)

(a) Explain briefly what a SAT solver is. (2p)

Solution

A SAT solver is a computer program for which takes as input a propositional logic formula, and tells us whether there is some assignment of true/false to each of the variables in the formula such that the formula is true.

(b) Many efficient SAT-solvers are available. How would you use property based testing, namely testing the pointwise equivalence of functions to test a SAT solver? Specify what you generate and when you detect that something is wrong. (2p)

Solution

We would generate random propositional logical formulas and feed them to the SAT solver we want to test, as well as to another SAT solver. If both say that the formula is satisfiable (not necessary the same assignment of variables) or not satisfiable, then we do not report that something is wrong. However, if one says that the formula is not satisfiable and the other says it is then we detect that something is wrong.

Assignment 6 Minimization (3p)

Suppose we have a method `f` which takes an array of characters as input, and suppose that this method computes the output incorrectly if the input contains two consequetive occurances of the letter `v`.

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

Solution

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

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

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

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

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

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

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

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

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

Assignment 7 Formal Specification (1) (3p)

The seL4 microkernel is a *verified* microkernel (a microkernel is the minimal core of an operating system).

→ Briefly explain what it means that the seL4 microkernel is verified. Use at least the following words in your answer: implementation, specification, refinement, executable specification, proof.

Solution
sel4 has three ingredients:

- A high level specification, which specifies what the system must do but leaves some choices open.
- An executable specification, which specifies what the system must do.
- An implementation.

When we say that sel4 is verified, we mean that there is a proof that each of these is a *refinement* of the previous: choices which were left open in at a higher level are filled in, but the rest of the behavior is the same.

Assignment 8 Formal Specification (2) (7p)

In this question you are going to specify and implement a method that takes two non-null arrays of the same length and "zips" them. This means that the method will return an array, as long as both input arrays together, where the elements alternately come from the first and the second input array.

For example, the result of running the method on the input arrays `[1,2,3,4]` and `[11,12,13,14]` will be a new array containing `[1,11,2,12,3,13,4,14]`.

The header of the method is as follows:

```
method zip(a : array<int>, b : array<int>) returns (c : array<int>)
requires ?
ensures ?
```

(a) Make the informal specification of `zip` formal by filling in the `requires` and `ensures` fields. (3p)

Solution

```
requires a != null && b != null && a.Length == b.Length
ensures c != null && c.Length == a.Length * 2 &&
         forall i : int :: 0 <= i < a.Length ==> c[2 * i] == a[i] && c
 [2 * i + 1] == b[i]
```

(b) Implement the `zip` 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

```
c := new int[a.Length * 2];
var i := 0;
while i < a.Length
invariant i <= a.Length &&
          forall j : int :: 0 <= j < i ==> c[2 * j] == a[j] && c[2 * j
 + 1] == b[j]
decreases a.Length - i
{
    c[2 * i] := a[i];
    c[2 * i + 1] := b[i];
    i := i + 1;
}
```

Assignment 9 (Formal Verification) (10p)

In this question, you are going to prove that a simple division method is correct using the weakest-precondition calculus. The following method implements the division of natural numbers:

```
method div(n : nat, d : nat) returns (q : nat, r : nat)
requires d > 0
ensures q * d + r == n && r < d
{
  r := n;
  q := 0;
  while r >= d
  invariant r + q * d == n
  decreases r
  {
      r := r - d;
      q := q + 1;
  }
}
```

The method only deals with whole numbers. The result q gives the number of times d fits in n and r is the remainder after division.

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

You can assume that any variable with type `nat` is always bigger or equal to 0.

Solution

Compute weakest postcondition :

`wp(r := n; q := 0; while r >= d ..., q * d + r == n && r < d)`

Apply seq rule (x2)

`wp( r := n, wp( q := 0, wp(while ..., q * d + r == n && r < d)))`

Compute `wp(while ..., q * d + r == n && r < d)` first

`wp(while (r >= d) (r + q * d == n) (n / d - q) r := r - d;q := q + 1, q * d + r == n && r < d)`

Which expands to (these should all hold):

1. Invariant holds before loop: `r + q * d == n`
2. Invariant maintained in loop: `r >= d && r + q * d == n ==>`
   `wp(r := r - d;q := q + 1, r + q * d == n)`

3. Invariant and loop fail implies postcondition:
   `!(r >= d) && r + q * d == n ==> q * d + r == n && r < d`
4. Decreases clause always positive : `q * d + r == n ==> r >= 0`
5. Iteration decreases : `r >= d && r + q * d == n ==>`
   `wp(tmp := r; r := r - d;q := q + 1, tmp > r)`

Simplify (2):

```
r >= d && r + q * d == n ==>
wp(r := r - d;q := q + 1, r + q * d == n)
```

Compute `wp(r := r - d;q := q + 1, r + q * d == n)`

Apply seq rule:
`wp(r := r - d, wp(q := q + 1, r + q * d == n)`
Apply assignment rule (x2)
`(r - d) + (q + 1) * d == n`

Plug in to (2):
`r >= d && r + q * d == n ==>`
`(r - d) + (q + 1) * d == n`
Simplify using `(r - d) + (q + 1) * d == n`math rewrite rules `(r - d) + (q + 1) * d == n` $\Leftrightarrow$ `r - d + q * d + d == n` $\Leftrightarrow$ `r + q * d == n`
This is an assumption we had, so reduces to true.

Simplify (3) :

`!(r >= d) && r + q * d == n ==> q * d + r == n && r < d`
Use `!(r >= d) = r < d`
`r < d && r + q * d == n ==> q * d + r == n && r < d`
Simplify using `a ==> a == True`
`True`

Simplify (4)

`r >= 0`
The type of r is nat, so True.

Simplify (5)

```
r >= d && r + q * d == n ==>
wp(tmp := r; r := r - d;q := q + 1, tmp > r
```

Compute:

```
wp(tmp := r; r := r - d;q := q + 1, tmp > r)
```

Seq rule (x2)

```
wp(tmp := r, wp ( r := r - d, wp (q := q + 1, tmp > r)))
```
Assignment rule (x 3)
```
r > r - d
```
Subtract `r` from both sides
```
0 > -d
d > 0
```
Now 2,3,5 reduce to true. So the
```
wp(while ..., q * d + r == n && r < d
```
= `d > 0 && q * d + r == n`

Plug back into:
```
wp( r := n, wp( q := 0 , wp(while ..., q * d + r == n)))
```
becomes:
```
wp( r := n, wp( q := 0 ,d > 0 && q * d + r == n))
```
Assignment (x2)
```
d > 0 && 0 * d + n == n
```
Use `0 * d == 0`
```
d > 0 && n == n
d > 0
```
So weakest precondition of program is `d > 0`.
Now check that our precondition `d > 0` implies:
`d > 0`. Which is obviously `True`