Testing, Debugging, and Verification exam
DIT082/TDA567

Day: 11 January 2016 Time: $14^{00} - 18^{00}$

Responsible:	Atze van der Ploeg
Results:	Will be published mid February
Extra aid:	Only dictionaries may be used. Other aids are *not* allowed!
Grade intervals:	**U**: 0 – 23p, **3**: 24 – 32p, **4**: 32 – 40p, **5**: 40 –46p, **G**: 24 – 39p, **VG**: 40 – 46p, **Max.** 46p.

Please observe the following:

- This exam has 14 numbered pages.
  Please check immediately that your copy is complete
- Answers must be given in English
- Use page numbering on your pages
- 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!

1 Testing

Assignment 1 Levels of testing (3p)

Crappy software Inc. has a launced a new social network. A few hours after launch, the users start complaining that there are messages being sent in their name, but that they did not send these messages themselves. After inspection, it turns out that a function `authenticate` authenticates anyone if they simply simply fill in more than 50 characters in the password field.

→ What is lowest level of testing detail at which a test could have caught this bug? Briefly motivate your answer. (3 points)

Solution

This bug could have been caught at the *unit* level. There could have for example been a unit test with a incorrect password of more than 50 characters for a login.

Assignment 2 Logic coverage (5p)

Consider the following piece of java code:

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

(a) Construct a minimal set of test-cases for the code snippet above, which satisfy *condition decision coverage* (3p)

Solution

{a = 0 , b = 1, c = 1}, {a = 1 , b = 0, c = 0}

(b) It is *impossible* to construct a set of test-cases for the code snippet above which satisfy *Modified Condition Decision Coverage*(MCDC). Explain why this is the case. (2p)

Solution

MCDC requires that conditions are independent. The conditions `a > b` and `b < a` are not independent.

Assignment 3 Branch coverage (4p)

Consider the following Java method:

```
/* merges two sorted lists

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 occurances in the
         output of i, is equal to the number of occurances in the left
         arrays of i plus the number of occurances 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;
  int ir = 0;
  int i = 0;
  while( i < res.length) {
    if (left[il] == right[ir]) {
      res[i] = left[il];
      res[i+1] = right[ir];
      i+=2; il += 1; ir+=1;
    } else if (left[il] < right[ir]){
       res[i] = left[il];
       i += 1; il += 1;
    } else { // right[ir] < left[il]
       res[i] = right[ir];
       i += 1; ir += 1;
    }
  }
  return res;
}
```

→ Write down one or more test cases, such that this/these test case(s) together satisfy *branch coverage*. Motivate your choice. (4 points)

Solution
For instance `merge({0,1,3},{0,2} == {0,1,2,3}`. No points subtracted if only inputs are given, but no output.

Assignment 4 Property based testing (3p)

A class for handling dates in a software project provides two methods:

```
// computes the year, when given the number of days since 1 Jan, 1980
// requires : days >= 0
// ensures  : output >= 1980
public static int yearFromDay(int days) ...

// computes the number of days since 1 Jan, 1980, on the first day of
 the given year
// requires : year >= 1980
   ensures  : output >= 0
public static int dayFromYear(int year)
```

You are in charge of testing these methods. To give more assurance than with unit tests, you want to use randomized (property based) testing.

→ Which property would you test? Describe the property which should hold if `yearFromDay` and `dayFromYear` are implemented correctly.

Solution

The property that should hold is: `forall x. yearFromDay(dayFromYear(x)) = x`

The other property `forall x. dayFromYear(yearFromDay(x)) = x` is *not* correct. For example, this does not hold for the 2nd day of 1980: `yearFromDay(1) == 1980` `dayFromYear(1980) == 0`

Assignment 5 Minimization using DDMin (6p)

(a) The `ddMin` algorithm computes a *1-minimal* failing input. Explain what a *1-minimal* failing input is. (2p)

Solution

A 1-minimal failing input is an input where if you remove any single character, the resulting input succeeds.

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 the substring `"foo"` (and otherwise computes the result correctly).

(b) Simulate a run of the `ddMin` algorithm and compute a 1-minimal failing input from the following initial failing input: `[b,a,l,f,o,o,b,a]`. Clearly state what happens at *each step* of the algorithm and what the final result is. (4p)

Solution

Start with granularity $n = 2$ and sequence `[b,a,l,f,o,o,b,a]`.

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

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

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

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

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

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

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

Assignment 6 Stateful property based-testing (7p)

A *multiset* (or *bag*) is a generalization of the concept of a set that, unlike a set, allows multiple instances of the multiset's elements. The *multiplicity* of an element is the number of instances of the element in a specific multiset.

For example, in the multiset $\{a, a, b\}$, a has multiplicity 2, and b has multiplicity 1.

Suppose we have mutable multiset of integers class in Java:

```
class MultiSet {

   // creates a new empty multiset
   MultiSet() { .. }

   // add an occurance of x to the multiset
   void add(int x) { ... }

   // remove an occurance of to the multiset
   void remove(int x) { ... }

   // gives the multiplicity of x
   int multiplicity(int x)

   // Gives an array, sorted in non-decreasing order, with all elements
   // in the multiset. Each element occurs the same number of times
   // in the array as it's multiplicity in the multiset.
   int[] allElements()
}
```

For example, consider the following method:

```
int[] test(){
  MultiSet x = new MultiSet();
  x.add(2);
  x.add(2);
  x.remove(3);
  x.add(1);
  return x.allElements();
}
```

This will return the array `[1,2,2]`.

Recall from the lectures that an algebraic property of a stateful object is two different methods operating on such a stateful object, such that there never is an (observable)

difference between executing one method or executing other method. For example for a *mutable set* of integers a property is:

```
void f(IntSet s, int y) {
  s.add(y);
  s.add(y);
}
```

==

```
void f(IntSet s, int y) {
  s.add(y)
}
```

(a) Think of two algebraic properties of the stateful multiset. (4p)

Solution
Example propeties are:

```
void method1a(MultiSet x, int y){
  x.add(y);
  x.remove(y);
}
==
void method1b(MultiSet x, int y){ }

void method2a(MultiSet x, int a, int b) {
  x.add(a);
  x.add(b);
}
==
void method2b(MultiSet x, int a, int b) {
  x.add(b);
  x.add(a);
}

void method3a(MultiSet x, int a) {
  if(x.multiplicity(a) == 0) {
    x.remove(a)
  }
}
==
void method3b(MultiSet x, int a) {
}

void method4a(MultiSet x, int a, int b) {
  x.remove(a);
  x.remove(b);
}
==
void method4b(MultiSet x, int a, int b) {
  x.remove(b);
  x.remove(a);
```

```
}
```

(b) Describe how you would use random testing to test this algebraic property. Include the following words in your answer: random, shared prefix, shared postfix, observe state. Don't forget to describe when the test succeeds or fails! (3p)

Solution
Generate a 2 random sequences of statements which manipulate a multiset using add and remove: the shared random prefix and postfix. Then create two programs: Both first create a new multiset, and then execute the shared random prefix. Then, one program executesone the side of the equivalence, and the other the other side. Afterwards, they both execture the shared random postfix and we observe their state by using allElements. If both arrays are (pointwise) equal, then the test succeeds, otherwise it fails.

Assignment 7 Formal Specification (8p)

For a user-authentication mechanism, someone has modeled a user as the following Dafny class:

```
class User{
  var userName : string;
  var password : string;
  var passwordHash : int;

  predicate hashIsCorrect()
  requires password != null && passwordHash != null
  {
    passwordHash == hashPassWord(password)
  }

  constructor (un : string, pw : string)
  requires un != null && pw != null
  ensures hashIsCorrect()
  {
    userName := un;
    password := pw;
    passwordHash := hashPassword(pw);
  }
}

function hashPashWord(x : string) : int
```

```
{ ... }
```

A user database is modelled as simply an array of users. We now want a predicate that check if an array of users is a valid user database. A valid user database is defined as an array where each element is non-null and the hash of each user is correct: the stored password hash is equal to the hash of the password.

(a) Write down the body of the (4p)
`predicate isValidUserDB(users : arr<User>) { ... }`
Use Dafny keywords and syntax in your answers. We do not subtract points for minor syntactical errors.

Solution

```
users != null && forall i :: 0 <= i < users.length ==> users[i] != null
  && hashIsCorrect(users[i])
```

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

```
method authenticate(users : arr<User>,
                    userName : string,
                    passwordHash : int) returns (authenticated : bool)
requires ?
ensures ?
```

Informally, the `authenticate` method takes a valid set of users, an non-null username and a password hash and returns true if the user is in the user database and the supplied password hash is correct, and false in all other cases.

(b) Write down the formal specifation of `authenticate`. In other words, fill in the `requires` and `ensures` clauses above. Use Dafny keywords and syntax in your answers. We do not subtract points for minor syntactical errors. (4p)

Solution

```
requires userName != null && isValidUserDB(users)
ensures authenticated <==> exists i :: 0 <= i < users.length
        && users[i].name == username
        && users[i].passwordHash == passwordHash
```

Assignment 8 (Formal Verification) (10p)

The nth power of 2, 2^n, can be defined in Dafny as follows:

```
function pow2(n : int) : int
requires n >= 0
{
  if n == 0 then 1 else 2 * pow2(n - 1)
}
```

The following small Dafny program does the following, given a number $x \geq 1$, it computes the number n such that $2^n \leq x < 2^{n+1}$.

```
method log2(x : int) returns (n : int, p : int)
requires x >= 1
ensures  n >= 0 && p == pow2(n) && 0 < p <= x < p * 2
{
  n := 0;
  p := 1;

  while (p * 2 <= x)
  invariant n >= 0 && p == pow2(n) && 0 < p <= x
  {
    n := n + 1;
    p := p * 2;
  }
}
```

(a) Prove partial correctness (no termination proof) for the above program. (5p)
You can assume that `p == pow2(n) ==> p * 2 == pow2(n+1)`.

Solution

1) Loop invariant holds on entry:

`P ==> wp(S, I)`

Which expands to:

`x >= 1 ==> wp(n := 0; p := 1, n >= 0 && p == pow2(n) && 0 < p <= x)`

Compute weakest precondition:

`wp(n := 0; p := 1, n >= 0 && p == pow2(n) && 0 < p <= x)`

Apply the Seq-rule:

`wp(n := 0, wp(p := 1, n >= 0 && p == pow2(n) && 0 < p <= x))`

Apply Assignment-rule:

```
wp(p := 1, 0 >= 0 && p == pow2(0) && 0 < p <= x)
```

Apply Assignment rule:

```
0 >= 0 && 1 == pow2(0) && 0 < 1 <= x
```

Simplify :

```
0 >= 0 && 1 == 1 && 0 < 1 <= x
```

```
1 <= x
```

Plug in weakest precondition:

```
x >= 1 ==> 1 <= x
```

Which is true by elemental algebra.

2) Prove loop invariant

```
E && I ==> wp(A,I)
```

Which expands to:

```
p * 2 <= x && n >= 0 && p == pow2(n) && 0 < p <= x ==>
wp(n := n + 1; p := p * 2, n >= 0 && p == pow2(n) && 0 < p <= x)
```

Compute weakest precondition:

```
wp(n := n + 1; p := p * 2, n >= 0 && p == pow2(n) && 0 < p <= x)
```

Apply the Seq-rule:

```
wp(n := n + 1, wp(p := p * 2, n >= 0 && p == pow2(n) && 0 < p <= x))
```

Apply Assignment:

```
wp(p := p * 2, n + 1 >= 0 && p == pow2(n + 1) && 0 < p <= x)
```

Apply Assignment:

```
n + 1 >= 0 && p * 2 == pow2(n + 1) && 0 < p * 2 <= x
```

Plug in weakest precondition:

```
p * 2 <= x && n >= 0 && p == pow2(n) && 0 < p <= x ==>
n + 1 >= 0 && p * 2 == pow2(n + 1) && 0 < p * 2 <= x
```

True by `p == pow2(n) ==> p * 2 == pow2(n+1)` and elemental algebra.

3) Loop invariant implies post condition

```
I && !E ==> wp(C, Q)
```

Which expands to:

```
n >= 0 && p == pow2(n) && p <= x && !(p * 2 <= x) ==>
wp( ,n >= 0 && p == pow2(n) && 0 < p <= x < p * 2)
```

By empty rule:

```
n >= 0 && p == pow2(n) && p <= x && !(p * 2 <= x) ==>
n >= 0 && p == pow2(n) && 0 < p <= x < p * 2
```

Simplify:

```
n >= 0 && p == pow2(n) && p <= x && p * 2 > x ==>
n >= 0 && p == pow2(n) && 0 < p <= x < p * 2
```

Simplify:

```
n >= 0 && p == pow2(n) && p <= x < p * 2 ==>
n >= 0 && p == pow2(n) && 0 < p <= x < p * 2
```

Which is true by `p ==> p == True`.

(b) What is a suitable variant (decreases clause) for the while loop in the above program? (1p)

Solution

```
x - p
```

(c) Prove termination of the while-loop for the above program using the variant from the previous sub-question. (4p)

Solution

1) Decreases clause is always $\geq$ 0.

```
I ==> D >= 0
```

Which expands to:

```
n >= 0 && p == pow2(n) && 0 < p <= x ==> x - p >= 0
```

Remove irrelavant bits:

```
0 < p <= x ==> x - p >= 0
```

Simplify.

```
0 < p <= x ==> x >= p
```

True by elemental algebra.

2) Decreases clause decreases each iteration.

```
E && I ==> wp(tmp := D ; S, tmp > D)
```

Which expands to:

```
p * 2 <= x && n >= 0 && p == pow2(n) && 0 < p <= x ==>
wp(tmp := x - p ; n := n + 1; p := p * 2, tmp > x - p)
```

Compute weakest precondition:

```
wp(tmp := x - p ; S, tmp > x - p)
```

By Seq rule (2x):

```
wp(tmp := x - p, wp (n := n + 1, wp(p := p * 2, tmp > x - p)))
```

By assignment rule (3x):

```
x - p > x - p * 2
```

Simplify

```
0 > - p 0 < p
```

Plug in:

```
p * 2 <= x && n >= 0 && p == pow2(n) && 0 < p <= x ==> p > 0
```

By `0 < p <==> p > 0`

(total 46p)