An introduction to formal specifications and JML

Proof obligations

Yves Ledru
Université Grenoble-1
Laboratoire d'Informatique de Grenoble
Yves.Ledru@imag.fr

2013

Avoiding impossible missions

- A specification may be unfeasible:

```
public class UnfeasibleSpec {

public int x;

//@ public invariant x > 0 && x < 0;

public UnfeasibleSpec(){
     x=1;
}
```

- It is impossible to find an implementation for the constructor!

Solutions of a quadratic equation

- In mathematics, finding the roots of a quadratic equation is a well-known problem:
- Equation : $ax^2 + bx + c = 0$
- Solutions : $(-b + \sqrt{b^2-4ac})/2a$
  and $(-b - \sqrt{b^2-4ac})/2a$
- Some of these equations have no solution. They correspond to an unsolvable problem.
- These unsolvable equations can be detected by checking that $b^2-4ac >= 0$

Can we detect that a specification is not implementable?

- Yes, this can be achieved by mathematical proofs.
- In this course, we will briefly consider two proof obligations associated to a JML class specification.

Let us consider the following class

```
public class C{
    public State s;
    //@ public invariant Inv(s);

    //@ requires pre_Op(s,i);
    //@ ensures post_Op(\old(s),s,i,\result);
    public Output Op(Input i){…}
```

- **s** corresponds to all state variables, and **i** to all input parameters of **Op**
- The invariant only refers to **s**
- The pre-condition refers to **s** (in the old state) and **i**
- The post-condition refers to **\old(s)**, **s**, **i** and **\result**

First proof obligation: state implementability

- The first proof obligation checks that there exists at least one value for **s** satisfying the invariant.

```
\exists State s; true; Inv(s)
```


Discharging the proof obligation

- It is rather easy to prove this first proof obligation:
- in order to prove an existential expression, you only need to provide one witness, i.e. one value of s satisfying Inv(s).
- Actually, the constructor of the class computes such a value.
- So if you can execute the constructor without raising an exception, you actually discharge this first obligation!

A simple example

```
public class SimpleCounter {
  public int c;
  //@ public invariant c >= 0;

  public SimpleCounter(){
    c = 0;
  }

  //@ requires c+x >= 0;
  //@ ensures c == \old(c)+x && \result == c;
  public int addToC(int x){
    c += x;
    return c;
  }
```

State s

Inv(s)

pre_Op(s,i)

post_Op(\old(s),s,i,\result)

Input i

Output

State implementability of SimpleCounter

- The proof obligation is

```
\exists State s; true; Inv(s)
```

- Replacing **`State s`** and **`Inv(s)`** by their instantiation results in:

```
\exists int c; true; c>=0;
```

- The constructor gives a potential witness:

```
public SimpleCounter(){c = 0;}
```

- Choosing c==0 proves the theorem

First conclusion

- The first proof obligation is actually trivial: there must exist one execution of the constructor whose resulting state fulfills the invariant!
- The proof obligation is established by a single test execution!

2nd proof obligation: Operation implementability

- The 2nd proof obligation checks that
  - If the operation started in an acceptable state (i.e., which fulfills the pre-condition and the invariant)
  - It is possible to find a result and a final state which fulfill the post-condition and the invariant
- In other words, the specification can be implemented.
- This proof obligation gives the actual semantics of the « contract », and checks that the contract makes sense.
- The difficulty is to prove that this is true for all acceptable initial states!

More formally

- The proof obligation is quantified forall initial states and inputs…

```
(\forall State old_s; Inv(old_s);
 (\forall Input i; pre_Op(old_s,i);

     (\exists State s; Inv(s);
      (\exists Output result; true;
          post_Op(old_s,s,i,result) ))))
```

- It requires the existence of at least one final state and result.

2nd proof obligation applied to addToC

```
(\forall int \old(c); \old(c) >= 0;
 (\forall int x; \old(c)+x >= 0;

     (\exists int c; c >= 0;
      (\exists int \result; true;
           c == \old(c)+x && \result == c ))))
```

- The proof is easy to establish:
  - The witnesses for c and result are both \old(c)+x (replacing c and result by this value fulfills the post-condition)
  - It remains to establish that \old(c)+x satisfies the invariant, which follows from the pre-condition!

The 2nd proof obligation in practice…

- It is rarely the case that the software engineer will compute the proof obligation, and discharge it by a proof…
- But the proof obligation may be used to help deduce the pre-condition, given a postcondition.

```
(\forall int \old(c); \old(c) >= 0;
 (\forall int x; ???FIND pre_Op(\old(c),x)???;

     (\exists int c; c >= 0;
      (\exists int \result; true;
           c == \old(c)+x && \result == c ))))
```

2nd proof obligation in practice (Ctd)

- The second proof obligation means that the pre-condition should be chosen to forbid inputs for which the operation breaks the invariant.
- A practical process is to
  - Start with a true precondition
  - Find examples which break the invariant
  - Design a new precondition
  - Replay the tests and check that they are blocked by the pre-condition!
- Designing the test before the pre-condition is a form of TDD (Test Driven Development)!

- Set **true** as pre-condition

```
public int c;
//@ public invariant c >= 0;

//@ requires true;
//@ ensures c == \old(c)+x
            && \result == c;
  public int addToC(int x){
    c += x; return c;}
```

- Design a test

```
SimpleCounter sc =
     new SimpleCounter();
sc.addToC(-1);
```

JMLInvariantError

- Design a precondition

```
public int c;
//@ public invariant c >= 0;

//@ requires c+x >= 0;
//@ ensures c == \old(c)+x
            && \result == c;
  public int addToC(int x){
    c += x; return c;}
```

- Replay the test

```
SimpleCounter sc =
     new SimpleCounter();
sc.addToC(-1);
```

JMLEntryPreconditionError

Conclusion

- Proof obligations give semantics to the contracts defined by the JML specification.
- They can be used in a formal development process, using proof tools… (not covered here)
- They also guide the testing process:
  - Write a test which successfully calls the constructor
  - Design tests for each operation potentially breaking the invariant, and add appropriate pre-conditions.