An introduction to formal specifications and JML

Operation specification

Yves Ledru
Université Grenoble-1
Laboratoire d'Informatique de Grenoble
Yves.Ledru@imag.fr
2013

Specifying an operation

• It is often the case that operations are specified by snapshots of the states before and after the operation

Before

After

• This way of specifying operations seems rather intuitive!
Design by Contract™ (Bertrand Meyer)

• The specification of an operation is seen as a contract which includes two conditions:
  – The post-condition describes a condition that the operation should establish in the final state
  – The pre-condition (optional) describes under which condition the operation should establish the post-condition
• If the pre-condition is not verified, the contract is not valid, and nothing is guaranteed in the post-state.

« Design by Contract » is a registred trademark of Eiffel Software in the USA.

Contracts with invariants

• Invariants are implicitly added to the pre- and post-conditions.

• In the following example, the post-condition refers to « contains » which itself returns a correct result only if the invariant is true.
**Postcondition: ensures**

- The postcondition uses keyword `ensures`
  ```java
  //@ ensures contains(v);
  public void insert(int v){…}
  
  //@ ensures !contains(v);
  public void delete(int v){…}
  ```

- Here, the post-condition of `insert(v)` guarantees that `v` will be an element of the set at the end of the operation!
- Moreover, the invariant will also be true!

**Contains(v)**

- `contains(v)` is a pure boolean method which checks that `v` is an element of the tree.
  ```java
  public /*@ pure */ boolean contains(int v){
    if (val == null) {return false;}
    else if (v == val.intValue()) {return true;}
    else if (v > val.intValue() && (rtree!= null))
      {return rtree.contains(v);}  
    else if (v < val.intValue() && (ltree!= null))
      {return ltree.contains(v);}  
    else {return false;}
  }
  ```

- It assumes that the tree is sorted, as described in the invariant.
Incomplete post-conditions

• As will be experimented in the exercises, the specifications of `insert` and `delete` are correct but incomplete.
• Additional properties will be specified later.

Pre-condition: requires

• Some operations cannot be applied in any initial state:
  – Because they would be unable to produce a result
  – Because they would end in an infinite loop or a run-time error.
• It is the case for `min()` and `max()` which only return a result if the set is not empty.
  ```java
  //@ requires !emptySet();
  public /*@ pure @*/ int max(){...}
  ```
• The pre-condition uses keyword `requires`
What means a pre-condition?

• It means that the operation expects the property to be true.
• If the property is false, nothing is guaranteed!

• For example, when refueling your car, it is forbidden to:
  – Smoke
  – Let your engine run
  – Give a phone call
• If you don’t obey these rules, it is not guaranteed that the refueling will proceed safely.

Inconclusive tests

• What do we learn from a test case which calls an operation without verifying its pre-condition?
• Nothing!
• For example, if you call max() on an empty set, you will get a run-time error!
• If you compile max() with jmlc, it will not execute the code, but raises an exception inside the pre-condition
Example of such a test

- The following test raises an EntryPrecondition error. The code of `max()` was not executed!

```java
@Test
public void testSequence_0() {
    SetAsTree s = new SetAsTree();
    int m = s.max();
}
```

```
1) testSequence_0(…)
org.jmlspecs.jmlrac.runtime.JMLEntryPreconditionError:
    by method SetAsTree.max regarding specifications at
    File "SetAsTree.java", line 111, character 29 when
        'this' is null
    at SetAsTree.checkPre$max$SetAsTree(SetAsTree.java:2713)
    at SetAsTree.max(SetAsTree.java:2807)
```

EntryPrecondition error

- Tests with an EntryPrecondition error are inconclusive, i.e. they neither reveal an error in the code nor in the specification.
- Inconclusive tests are simply useless test cases!
- For example, if you test your car under water, you will learn nothing about its conformance to its specification!
**InternalPrecondition error**

- Precondition errors may also result from an erroneous use of `max()` in the code of another operation (e.g. `delete`).

```java
public void delete(int v){
    ... if (!ltree.emptySet()){
        int newVal = ltree.max();
        // take the largest element of the left tree
        ltree.delete(newVal);
        // remove it from the left tree
        ...}
```

- If the programmer forgets to test that the ltree is not empty, this code may result in a false precondition for `max()`.
- In this case, it corresponds to a programming fault!
- Therefore JML will return an `InternalPrecondition` error.

**Declarative specification**

- The pre- and post-condition specify what should be done, but not how to do it.
- The code is responsible to fulfill the specification.
- Several codes are acceptable, provided that the pre/post conditions are verified.
- This allows evolutions and optimisations of the code, provided they conform to the specification.
Towards more complete post-conditions

Incomplete post-conditions

• The current post-condition of `insert` and `delete` does only express that v is or is not in the tree.
• But it does not tell what happened to the other values in the tree!
• So the following trivial implementation of `insert` conforms to the specification!

```java
//@ ensures contains(v);
public void insert(int v){
    val = new Integer(v);
    ltree = null;
    rtree = null;
}
```
An informal but more complete specification

- A more complete specification:
  « The set contains v and the other elements of the set remain in the set »
- This specification expresses a property which links two states:
  - The initial state where the operation started
  - The final state where the operation ended
- The post-condition is expressed in the final state.
- We need a construct to refer to the initial state in the post-condition!

\old(expression)

- The JML \old construct returns the value of its expression in the initial state.
- \old may only be used in post-conditions.
**Evolution of the number of elements (1)**

- `insert` and `delete` modify the number of elements in the tree.
- Let us define a `size()` method:

  ```java
  public /*@ pure @*/ int size(){
      int size = 1;
      if (val==null){return 0;}
      else {
          if (ltree != null){size+=ltree.size();};
          if (rtree != null){size+=rtree.size();};
          return size;
      }
  }
  ```

**Evolution of the number of elements (2)**

- Inserting an element will increase the number of elements by one or leave it unchanged (if the element was already in the tree).
- We can thus express the following post-condition for `insert`:

  ```java
  //@ ensures contains(v);
 /*@ ensures size() == \old(size())+1 @*]
  public void insert(int v){…}
  ```

- This condition forbids some trivial implementations of the specification.
Evolution of the number of elements (3)

• The post-condition may even be more precise.

```java
//@ ensures contains(v);
//@ ensures \old(contains(v)) ==> size() == \old(size());
//@ ensures \old(!contains(v)) ==> size() == \old(size())+1;
public void insert(int v){…}
```

• \( A \implies B \) means that \( A \) implies \( B \),
  it is equivalent to \( \neg A \lor B \)

Limits of the current specification

• The post-condition simply constrains the number of elements in the tree,…
• …, but not the values of these elements.

• The following specification keeps the values of old elements.
Converting to another set...

- The following method converts the tree to a HashSet

```java
public /*@ pure */ HashSet toHashSet()
{  
    HashSet hs = new HashSet();  
    if (val==null){return hs;}  
    else { hs.add(val);  
        if (ltree != null){hs.addAll(ltree.toHashSet());};  
        if (rtree != null){hs.addAll(rtree.toHashSet());};  
        return hs;  
    }
}
```

Post-condition based on HashSet

- The following post-condition ensures that
  - The new value is in the set
  - The old values remain in the set
  - No other value is added to the set

```java
//@ ensures contains(v);
//@ ensures \old(contains(v)) ==> size() == \old(size());
//@ ensures \old(!contains(v))==> size() == \old(size())+1;
//@ ensures toHashSet().containsAll(\old(toHashSet()));
public void insert(int v){...}
```

- This specification is complete, but relies on an equivalent data structure (HashSet).
Computing the sum of elements

- The value of each element of the tree contributes to the sum of its elements.

```java
public /*@ pure @*/ int sum(){
    if (val==null){return 0;}
    else { int s = val.intValue();
        if (ltree != null){s+=ltree.sum();};
        if (rtree != null){s+=rtree.sum();};
        return s;
    }
}
```

Postcondition based on the sum

- The following specification is not complete, but rather constraining and does not rely on an equivalent data structure.

```java
//@ ensures contains(v);
//@ ensures \old(contains(v)) ==> size() == \old(size());
//@ ensures \old(!contains(v))==> size() == \old(size())+1;
//@ ensures \old(contains(v)) ==> sum() == \old(sum());
//@ ensures \old(!contains(v))==> sum() == \old(sum())+v;
public void insert(int v){...}
```
\texttt{\texttt{old} and references to objects}

- Take care that \texttt{old} takes an expression as argument.
- If the expression is the name of an object, it returns the address of the object, and not its value!
- The address of the object is the same in the initial and final state: \texttt{old(s)==s}
- If the postcondition needs to compare the value of the object, make sure you return its value, and not its address!

\texttt{Example of wrong use of old}

```java
public static StringBuffer s = new StringBuffer("abc");

public static void duplicate(){
    s.append(s);
}

- Calling \texttt{duplicate()} changes s to « abcabc »
- Correct specification:
  ```java
  //@ ensures s.length() == \texttt{old(s).length()}*2;
  ```
- Incorrect specification:
  ```java
  //@ ensures s.length() == \texttt{old(s).length()}*2;
  ```

Exception … JMLInternalNormalPostconditionError: by method DuplicateStringBuffer.duplicate … when \texttt{old(s)} is abcabc
Specifying methods with a return value

• The post-condition specifies the final state, but also the result of a method.
• result can be used in the post-condition to refer to the result of the method.
• For example, here is a specification of min (taking into account that min is computed recursively)

```java
//@ requires !emptySet();
//@ ensures contains(result) && result <= val.intValue();
//@ ensures ltree != null ==> result != val.intValue();
public /*@ pure @*/ int min(){…}
```

• The third line forbids the trivial case where the value of the root is returned.

Some JML abbreviations

• The following notations can be used in JML assertions:
  - ==> implication
  - <= inverse implication
  - <==> if and only if
  - <=!=> not if and only if

• Other keywords can be found in the JML reference manual.
Quantifiers (1)

• JML provides several quantifiers for assertions.

```java
public int[] table = {1,2,3,4,5};
/*@ public invariant
@ (\forall int i; 0<=i && i<table.length; table[i] > 0);
@*/
```

• They can be used to express a property on several objects.
• Without quantifiers, this would be expressed in the code of an iterative Java method.

Quantifiers (2)

• A quantified expression is of the form:
  • (\quantifier declaration ; boolean expr ; boolean expr )
    – The quantifier is one of (\forall, \exists, \num_of)
    – The declaration introduces the quantified variable
    – The first boolean expression constrains the range of the quantified variable
    – The second boolean expression is evaluated on all elements of the range
Non-executable quantification

- Please note that the range expression must be of the form
  \[ A<i \land i<B \] (< may be replaced by \(\leq\))
  or \(\text{JMLCollection}\_\text{has}(i)\)
  to be executable.
- Other boolean expressions used as range expressions are
  syntactically correct but will not be used by the run-time
  assertion checks.
- Example of non executable quantification

```java
//@ public invariant (\forall int i; true; i!=0);
```

File "Quantifiers.java", line 12, character 30 warning:
This quantifier is not executable.

Other examples of quantifiers

- Existential quantifier

```java
/*@ ensures
  @ (\exists int i; 0 <= i \land i < table.length;
      @
      table[i] == \result);
  @*/
  public int choose_one(){...}
```

- Number of elements satifying a property

```java
/*@ ensures
  @ \result == (\num_of int i; 0 <= i \land i < table.length;
    @
    table[i]%2 == 0);
  @*/
  public int nb_even(){...}
```
Other JML keywords

• There exist other JML keywords for boolean expressions and quantifiers.
• They will not be used in this course.
• Please refer to the JML Reference manual for a more complete information.

Visibility of assertions and variables
Visibility of variables and assertions

- In Java, variables and methods can be declared as public, protected or private.
- JML specifications are usually public but may refer to private variables.
- Such private variables must be declared as /*@ spec_public @*/.

Private invariants

- There are not many cases where invariants should be kept private.
- The main case is
  - when there is an inheritance relation between two classes
  - Private invariants are not inherited
  - So, choose private invariants to express a property on public variables that should not be inherited.
Example of spec_public variable

```java
public class Visibility {
    private /*@ spec_public */ int x;
    //@ public invariant x > 0;

    //@ requires v > 0;
    //@ ensures x == v;
    public Visibility(int v){
        x = v;
    }

    //@ ensures \result == x;
    public int getX(){
        return x;
    }
}
```

x is declared private to restrict its access to getters and setters.

x is used in the public specification of public operations. It must thus be visible in JML (spec_public).