Advanced JML

Erik Poll

Radboud University Nijmegen
Remember the core JML keywords were

- requires
- ensures
- signals
- invariant
- non_null
- pure
- \old, \forall, \result
More advanced JML features

- Visibility
- Specification inheritance, ensuring behavioural subtyping
- `normal behavior`, `exceptional behavior`
- `model fields`
- `ghost fields`
Visibility

JML imposes visibility rules similar to Java, eg.

```java
public class Bag{
    ...
    private int n;

    //@ requires n > 0;
    public int extractMin(){  ...  }
```

is not type-correct, because public method `extractMin` refers to `private` field `n`.
Visibility

```java
public int pub;  private int priv;

//@ requires i <= pub;
public void pub1 (int i) { ... }

//@ requires i <= pub && i <= priv;
private void priv1 (int i) ...

//@ requires i <= pub && i <= priv; // WRONG !!
public void pub2(int i) { ... }
```
Visibility: spec_public

Keyword `spec_public` loosens visibility for specs. Private `spec_public` fields are allowed in public specs, e.g.:

```java
public class Bag{
    ... 
    private /*@ spec_public @*/ int n;

    //@ requires n > 0;
    public int extractMin(){ ... }
}
```

Exposing private details can be ugly, of course. A nicer, but more advanced alternative is to use public model fields to represent (abstract away from) private implementation details.
signals and normal_behavior

Exceptions are allowed by default, i.e. the default signals clause is

```java
signals (Exception) true;
```

To rule them out, add an explicit

```java
signals (Exception) false;
```

or use the keyword `normal_behavior`

```java
/*@ normal_behavior
  requires ...
  ensures ...
  @*/
```
exceptional_behavior

normal_behavior has implicit signals(Exception) false

exceptional_behavior has implicit ensures false

Eg.

/*@ normal_behavior
    requires amount <= balance;
    ensures ...

also

    exceptional_behavior
    requires amount > balance
    signals (BankAccountException e) ...

@*/

public int debit(int amount) {
    ... }

JML – p.8/23
signals vs exceptional_behavior

Beware of the difference between

(1) if P holds, then SomeException is thrown

and

(2) if SomeException is thrown, then P holds

(1) can be expressed with exceptional_behavior, 
(2) with a signals clause.
Suppose \texttt{Child} extends \texttt{Parent}.

- **Behavioural subtyping** = objects from subclass \texttt{Child} “behave like” objects from superclass \texttt{Parent}

- **Principle of substitutivity** [Liskov]: code will behave “as expected” if we provide an \texttt{Child} object where a \texttt{Parent} object was expected.
Behavioural subtyping can be enforced by insisting that

- invariant in subclass is **stronger** than invariant in superclass
- for every method,
  - precondition in subclass is **weaker (!)** than precondition is superclass
  - postcondition in subclass is **stronger** than postcondition is superclass

JML achieves this using **specification inheritance**: any child class **inherits** the specification of its parent.
Invariants are inherited in subclasses. Eg.

```java
class Parent {
    ...
    //@ invariant invParent;
    ...
}

class Child extends Parent {
    ...
    //@ invariant invChild;
    ...
}

the invariant for Child is invChild && invParent
```
Keyword *also* indicates there are inherited specs.
Specification inheritance for methods specs

Method \texttt{m} in \texttt{Child} also has to meet the spec given in \texttt{Parent} class. So the complete spec for \texttt{Child} is

\begin{verbatim}
class Child extends Parent {

//@ requires i \geq 0;
//@ ensures \result \geq i;
//@ also
//@ requires i \leq 0
//@ ensures \result \leq i;

int m(int i){ ... }
}
\end{verbatim}

What can result of \texttt{m(0)} be?
This is equivalent with

```java
class Child extends Parent {

   //@ requires i <= 0 || i >= 0;
   //@ ensures \old(i) >= 0 ==> \result >= i;
   //@ ensures \old(i) <= 0 ==> \result <= i;
   int m(int i) { ... }
}
```
Ghost fields

Sometimes it is convenient to introduce an extra field, only for the purpose of specification (aka auxiliary variable).

A **ghost** field is like a normal field, except that it can only be used in specifications.

A special **set** command can be used to assign a value to a **ghost** field.
Suppose the informal spec of

```java
class SimpleProtocol {

    void startProtocol() { ... }

    void endProtocol() { ... }
}
```

says that `endProtocol()` must only be invoked after `startProtocol()`, and vice versa.

This can be expressed using a `ghost` field, to represent the “state” of the object.
class SimpleProtocol {
    //@ boolean ghost started;
    //@ requires !started;
    //@ ensures started;
    void startProtocol() {
        ...
        //@ set started = true;
    }
    //@ requires started;
    //@ ensures !started;
    void endProtocol() {
        ...
        //@ set started = false;
    }
}
Maybe the object has some internal state that records if protocols is in progress, e.g.

```java
class SimpleProtocol {
    //@ private ProtocolStack st;
    ...
    void startProtocol() {
        ...
        st = new ProtocolStack(...);
        ...
    }

    void endProtocol() {
        ...
        st = null;
        ...
    }
}
```
There may be correspondence between the ghost field and some other field(s), eg.

class SimpleProtocol {
    //@ private ProtocolStack st;
    //@ boolean ghost started;
    //@ invariant started <=> (st !=null);
    //@ requires !started;
    //@ ensures started;
    void startProtocol() { ... }

    //@ requires started;
    //@ ensures !started;
    void endProtocol() { ... }
}
Ghost fields - example

We could now get rid of the ghost field, and write

class SimpleProtocol {
    //@ private ProtocolStack st;

    //@ requires !(st!=null);
    //@ ensures (st!=null);
    void startProtocol() { ... }  

    //@ requires (st!=null);
    //@ ensures !(st!=null);
    void endProtocol() { ... }

but this is ugly...

Also, st must now be spec public.
Model fields - example

Solution: use a `model` field

```java
class SimpleProtocol {
    //@ private ProtocolStack st;

    //@ boolean model started;
    //@ represents started <-- (st!=null);

    //@ requires !started;
    //@ ensures started;
    void startProtocol() {
        ... }

    //@ requires started;
    //@ ensures !started;
    void endProtocol() {
        ... }
}
```
Model vs ghost fields

Difference between **ghost** and **model** is maybe confusing! Both exist only in JML specification, and not in the code.

- **Ghost**
  - Ghost field is like a normal field.
  - You can assign to it, using `set`, in JML annotations.

- **Model**
  - Model field is an abstract field.
  - Model field is a convenient abbreviation.
  - You cannot assign to it.
  - Model field changes its value whenever the representation changes.

Model field is like ‘abstract value’ for ADT (algebraic data type), represent clause is like ‘representation function’.