Desugaring JML method specifications

A.D. Raghavan and G.T. Leavens

Julien Charles

EVEREST

Tuesday, july the 11th 2006

Disclaimer

- Really informal
  (previously : *Exploiting purity for atomicity*)
- Back to basics, a 'classical' article
- Mariela : "It's only about tool smithing"
  $\longrightarrow$ Desugaring is a key idea in JML
  (behavior + refinement)

General Plan

1. JML
2. Desugaring
3. Next steps

JML

The Java Modelling Language (JML) is an annotation language for Java programs

In JML you can express loop invariants, assertions method's pre and post conditions.

JML has a really large syntax so every tool (except maybe JMLrac) implements only a portion of its syntax.

The main concept (again) : modelling by specification cases (behaviors)

You can also specify only parts of your specifications and refine them.

Some JML constructions (1)

- Expressions used in annotations are special boolean expressions :
  - \forall
  - \exists
  - \old
  - methods if they are pure
- Loop invariants, termination :
  - loop_invariant
  - decreases
  - loop_modifies
- Assertions with the assert construct

Some JML constructions (2)

- specification variables
  - ghost v ;
  - set v = value ;
  - let v...
- Methods :
  - requires
  - ensures
  - modifies
- Behaviors
  - normal_behavior
  - exceptional_behavior

Method specifications

Method specs can be :

- private
- protected
- public

Different cases can be specified using the also keyword

Specifications can be combined : the and keyword

A typical method specification

```
/*@ public normal_behavior
  @     requires !empty();
  @     ensures \result == theElems.header();
  @ also
  @ public exceptional_behavior
  @     requires empty();
  @     signals (EmptyException e) true;
  @*/
public Object top() throws EmptyException;
```

A 5 step way to desugaring

1. Nested specification
2. Normal and exceptional specifications
3. Inheritance and refinement
4. Multiple requires clauses
5. The also combinations

Nested specifications (1)

Specs can be factored :

```
/*@ requires x > 0;
  @ {|
  @     requires x % 2 == 1;
  @     ensures \result == 3*x + 1;
  @ also
  @     requires x % 2 == 0;
  @     ensures \result == x / 2;
  @ |}
  @*/
int hailstone(int x);
```

Nested specifications (2)

Desugared into :

```
/*@    requires x > 0 ;
  @    requires x % 2 == 1 ;
  @    ensures \result == 3*x + 1 ;
  @ also
  @    requires x > 0 ;
  @    requires x % 2 == 0 ;
  @    ensures \result == x / 2 ;
  @*/
int hailstone(int x) ;
```

Desugaring behaviors (1)

(the slide from the 1st part ;))

```
/*@ public normal_behavior
  @     requires !empty() ;
  @     ensures \result == theElems.header() ;
  @ also
  @ public exceptional_behavior
  @     requires empty() ;
  @     signals (EmptyException e) true ;
  @*/
public Object top() throws EmptyException ;
```

Desugaring behaviors (2)

```
/*@ public behavior
  @     requires !empty();
  @     ensures \result == theElems.header();
  @     signals (Exception e) false;
  @ also
  @     requires empty();
  @     ensures false;
  @     signals (EmptyException e) true;
  @*/
public Object top() throws EmptyException;
```

Refinement and inheritance

Refinement : every specification are combined
Inheritance : only the public specifications

also (a whiteboard example)
and (a whiteboard example too)

Requires / signals

- Multiple requires :

  requires a ;

  requires b ;

  turns to :

  requires a && b ;

- signals are standardized :

  signals (EmptyException e) P(e) ;

  turns to :

  signals (Exception e) (e instanceof EmptyException) && P(e) ;

The also desugaring (1)

The final step : desugaring also.

```
  requires r1 ;
  modifies m1 ;
  ensures e1
  signals (Exception e) s1 ;
also
  requires r2 ;
  modifies m2 ;
  ensures e2
  signals (Exception e) s2 ;
```

The also desugaring (2)

requires $r_1 \mid\mid r_2$;
modifies (m_1 if r_1) &&
    (m_2 if r_2) ;
ensures ($r_1 ==> e_1$) &&
    ($r_2 ==> e_2$) ;
signals (Exception e)
    ($r_1 ==> s_1$) &&
    ($r_2 ==> s_2$) ;

Conclusion

JML has lots of keyword
$\longrightarrow$ Desugaring limits the general explosion
Behavior would be some hard to manage annotations

Unclear in this paper :
Model vs. Ghost

Other desugarings ?

- Model variables (P. Müller et al)
- Some of the JML keywords into specification methods call (aka. model methods call)

Model variables

- Model variables are specification variables associated with a program variable.
- The association is done through a translation function.
- In JML it has the following syntax :
  - //@ model MyType modVar ;
  - //@ represents modVar $\leftarrow$ P(progVar)
- You cannot do a 'set' with these variables, i.e. you cannot assign them.
  $\rightarrow$ The model variable is considered to be modified at the same time the program variable it represents

POs with model

For the post-condition of each method which is using progVar :

- Model variables are desugared by adding the condition :
  $\exists$ v, v $=$ P(progVar)
- We then replace each occurence of modVar in the post-condition of the method by v