Métodos Formais em Engenharia de Software

JML: beyond the basics

José Carlos Bacelar Almeida

Departamento de Informática
Universidade do Minho

MI/MEI 2008/2009

Talk Outline

- JML: beyond the basics
  - visibility
  - most used clauses...
- Problem: Aliasing
  - aliasing of arguments
  - reference escaping
- Specification Inheritance
  - inheritance rules
  - common mistakes
- Abstraction in Specifications
  - data groups
  - model fields and representation

...these slides were prepared by adopting/adapting "teaching material" from the JML and ESC/Java2 sites.

Visibility for Specifications

- JML adopts the same visibility rules from Java (private, protected, public)

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

```
public class XXX {
  public int x; private int y; private /*@ spec_public @*/ int z;
...
  //@ requires x>z;
  public int fff(){ ... }

  //@ requires y>0;
  private int ggg(){ ... }
```


Multiple Specification Cases

- It is often easier to split complex specifications in multiple cases

```
private /*@ spec_public @*/ int age;

/*@   requires 0 <= a && a <= 150;
  @   assignable age;
  @   ensures age == a;
  @ also
  @   requires a < 0;
  @   assignable \nothing;
  @   ensures age == \old(age);
  @*/
  public void setAge(int a)
  { if (0 <= a && a <= 150) { age = a; } }
```

Assertion Semantics

- A JML assertion is taken to be valid if and only if:
  - does not cause an exception to be raised
  - returns value "true"
- Exceptions should be avoided by the specifier and tools are encouraged to warn users when they detect them.
- To avoid exceptions during evaluation:
  - practice good Java coding habits
  - write specifications that prevent such exception:
    - use of short-circuit Java operators ("&&" and "||")
    - multiple clauses

```
/*@ requires field != null;
  @ requires field.data > 0;
  @ ensures ...;
  @*/
  public void setField(MyObject field)
  { ... }
```


Nested Specification Groups

- ...more about multiple specification cases...
- first "requires" clause distributes with each case

```
/*@ requires 0 < n;
  @ {|
  @   requires n % 2 != 0;
  @   ensures \result == (3*n+1)/2;
  @  also
  @   requires n % 2 == 0;
  @   ensures \result == n/2;
  @ |}
  @*/
public static /*@ pure @*/ int h(int n) {
...
}
```

Ghost variables

- Ghost fields behave like normal Java fields, but only affects specifications (are ignored by the Java compiler...)

```
public class XXX {
  //@ public ghost boolean started = false;
...
  //@ require !started
  public void start() {
    //@ set started = true;
    ...
  }
 //@ require started
  public void end() {
    //@ set started = false;
    ...
  }
}
```


History Constraints

- Invariants specify state properties:

```
public class MonotoneCounter {
    private /*@ spec_public @*/ int val;
    //@ invariant val>=0;
  ...
    //@ ensures val==\old(val)+1;
    public void tickCounter() {
        val ++;
    }
}
```

- Sometimes, it is convenient to specify the admissible state transformations:

```
public class MonotoneCounter {
    private /*@ spec_public @*/ int val;
    //@ initially val==0;
    //@ constraint val>\old(val);
  ...
    public void tickCounter() {
        val ++;
    }
}
```

Frame conditions

- Frame conditions (assignable clause) restrict possible side-effects of the methods (i.e. "where" the method is allowed to make changes)

```
/*@ requires amount >= 0;
    assignable balance;     //balance is an instance variable...
    ensures balance == \old(balance)-amount;
  @*/
public void debit(int amount) {
  balance = balance - amount;
}
```

- They are a crucial ingredient when reasoning about programs...

```
...
//@ assume name!=null;
debit(50);
// ??? name!=null ???
...
```

- Default assignable clause: assignable \everything.
- Pure methods are implicitly assignable \nothing.
- Synonyms: modifies, modifiable; ensures \only_assigned(gender)

Loop Invariants

- When reasoning about cycles, we need to annotate them with invariants and variants.
- JML clauses:

```
int f = 1 ;
int i = 1 ;
/*@ loop_invariant i <= n &&
                   f == (\product int j ; 1 <= j && j <= i ; j ) ;
    decreases n-i;
  @*/
while ( i < n ) {
  i = i + 1 ;
  f = f * i ;
}
```

- JML tools often translate these to "appropriate" assert/assume clauses
- ...but appropriateness in this context does not always mean "sound"...

Exceptional Behaviour

- The "ensures" clause characterises only the "normal" control flow of methods.
- To specify properties under "exceptional" results, the signals clause can be used:

```
/*@ requires amount >= 0;
    signals (BankAccountException e) balance==\old(balance);
    ensures balance == \old(balance)-amount;
  @*/
public void debit(int amount) {
  balance = balance - amount;
}
```

  - meaning: if "BankAccountException" is thrown, balance remains unchanged.
- By default, exceptions (declared as "throwable") are allowed (the default clause is "signals (Exception e) true;")
- To disallow them, an explicit "signals (Exception e) false;" must be given.

Lightweight vs. Heavyweight Specifications

- In fact, JML distinguishes between two forms of specifications:
  - lightweight specifications: specify "normal behaviour" (possibly with "\signal" clauses)
  - heavyweight specification: separate "normal" and "exceptional" behaviour specification.

```
/*@ normal_behavior
      requires amount <= balance;
      ensures ...
   also
    exceptional_behavior
      requires amount > balance
      signals (BankException e) ...
  @*/
  public int debit(int amount) throws BankException
  { ... }
```

- "normal_behavior" has an implicit "signals (Exception e) false;"
- "exceptional_behavior" has an implicit "ensures false;"

- "signals_only E_1, ..., E_n" limits the set of allowed exceptions.
- "signals_only E_1, ..., E_n" is a synonymous for:

```
signals (Exception e) e instanceof E1
                      || ...
                      || e instanceof En;
```

Warning: exceptional specifications are easy to get wrong!

Aliasing

- Does the following method satisfy its contract?

```
public class Counter {
    private /*@ spec_public @*/ int val;
    //@ invariant val>=0;
    ...
    //@ ensures val==\old(val)+c.val;
    public void addCounter(/*@ non_null @*/ Counter c) {
        val += c.val;
    }
}
```

Aliasing

- Does the following method satisfy its contract?

```
public class Counter {
    private /*@ spec_public @*/ int val;
    //@ invariant val>=0;
    ...
    //@ ensures val==\old(val)+c.val;
    public void addCounter(/*@ non_null @*/ Counter c) {
        val += c.val;
    }
}
```

- ...in fact, ESC/Java warns about a post-condition violation...
- But this actually anticipates deeper concerns when aliasing comes into play:

  "Modular verification is not possible in the presence of aliasing"

- ...and Java doesn't constrain "reference leaks" in methods...

```
public class MyClass {
    private /*@ spec_public @*/ int a[];
    //@ invariant (\forall int i; 0<=i && i<N; a[i]>=0);
    ...
    //@ ensures result==a;
    public int[] getArray() { return a; }
}
```


- Solution #1 (ESC/Java)
  - explicitly handles the ghost "owner" field (declared in the Object class)

- Solution #2 (jmlc)
  - Universes Type System (P. Müller) - statically enforces the "owner as modifier" discipline
    - small overhead on the programmer (rep, peer and readonly type annotations)
      - rep - owner is the receiver;
      - peer - same owner as the receiver.
    - but excludes some unproblematic situations...

Non-Functional Requirements

- JML supports some non-functional requirements
  - time and space constraints (\duration, \space, \working_space operators)
  - concurrency (\when, \lockset, ...)
  - ...
- Moreover, clever uses of ghost variables often allow for sound encodings of some of these requirements (specification patterns)
  - Method-call sequencing constraints
  - Non-interference
  - ...
- And, and course, a new extension can always be proposed...
  - Architectural constraints (embedded ACL in JML),
  - ...

Exercises:

http://www.cs.ru.nl/~erikpoll/Teaching/JML/taxpayer.html

Inheritance of Specifications

- Inheritance of specifications occur when:
  - a class extends another (sub-classing);
  - implementation of interfaces.
- All the behaviour specifications are inherithed:
  - invariants, initially and history constraints;
  - methods pre and post-conditions (actually, all the specification cases)

```
class Parent {
private /*@ spec_public @*/ int age;
//@ invariant age <= 150;
...
}
class Child extends Parent {
//@ invariant age <= 18;
...
}
```


behavioural sub-typing

- Behavioural subtyping:
  - objects from subclass Child "behave like" objects from superclass Parent.
- Principle of substitutivity [Liskov]:

  code will behave "as expected" if we provide a Child object where a Parent object was expected.
- Consider the following example:

```
class Parent {
//@ requires i >= 0;
//@ ensures \result >= i;
int m(int i){ ... }
}
class Child extends Parent {
//@ also
//@   requires i <= 10;
//@   ensures \result <= i;
int m(int i){ ... }
}
```

behavioural sub-typing

- We might expect that method "m()" on the Child class "specialises" the pre-condition...
- ... but the resultant specification is:

```
class Child extends Parent {
/*@   requires i >= 0;
  @   ensures \result >= i;
  @ also
  @   requires i <= 10;
  @   ensures \result <= i;
  @*/
int m(int i){ ... }
}
```

- Which specifies a "special case" (it does not override the inherited one):

```
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){ ... }
}
```


behavioural sub-typing

- When we are interested in characterising the exact behaviour of methods acting on objects with a specific dynamic type, we can do something like:

```
public class Object {
/*@ ensures (this == o) ==> \result;
  @ ensures \typeof(this) == \type(Object)
  @                   ==> (\result == (this==o));
  @*/
public boolean equals(Object o);
}
```

```
/*@ requires p instanceof Doctor
  @   || p instanceof Nurse; @*/
public boolean isHead(final Staff p) {
  if (p instanceof Doctor) {
    Doctor doc = (Doctor) p;
    return doc.getTitle().startsWith("Head");
  } else {
    Nurse nrs = (Nurse) p;
    return nrs.isChief();
  }
}
```

Datagroups

- Assignable clauses are crucial for reasoning about specifications, but they tend to:
  - expose implementation details:
  - become very long:

```
public class Timer{
  /*@ spec_public @*/ int time_hrs, time_mins, time_secs;
  /*@ spec_public @*/ int alarm_hrs, alarm_mins, alarm_secs;

  //@ assignable time_hrs, time_mins, time_secs;
  public void tick() { ... }

  //@ assignable alarm_hrs, alarm_mins, alarm_secs ;
  public void setAlarm(int hrs, int mins, int secs) { ... }
}
```

- Datagroups provide an abstraction mechanism for assignable clauses.

Datagroups

```
public class Timer{
  //@ public model JMLDatagroup time, alarm;
  int time_hrs, time_mins, time_secs; //@ in time;
  int alarm_hrs, alarm_mins, alarm_secs; //@ in alarm;

  //@ assignable time;
  public void tick() { ... }

  //@ assignable alarm;
  public void setAlarm(int hrs, int mins, int secs) { ... }
}
```

- Datagroups can be nested

```
//@ public model JMLDatagroup time, alarm;//@ in objectState;
```

- There's a default datagroup objectState defined in "Object.java"
- It's good practice to declare that all instance fields are in objectState

Abstraction in Specifications

- Model fields:
  - used for specification purposes;
  - "represent" concrete fields.

```
public interface Gendered {
  //@ public model instance String gender;

  //@ ensures \result == gender.equals("female");
  public /*@ pure @*/ boolean isFemale();
}
```

obs: "instance" modifier overrides default Java's "static" modifier for interface fields.

- Actual fields can be "abstracted by" the model field:

```
public class Animal implements Gendered {
  protected boolean gen; //@ in gender;
  /*@ protected represents
    @   gender <- (gen ? "female" : "male");
    @*/
  public /*@ pure @*/ boolean isFemale() {
    return gen;
   }
```


Abstract types for Specifications

- JML defines a rich set of data-types often used during specifications:
  - Object and Value Collections (Set, Bag, Collection, ...)
  - Maps, Relations, ...
- Access to these types requires:

```
//@ model import org.jmlspecs.models.*;
```

- In general, these are immutable "pure" Java objects (suitable for using in specifications).
- ...they shall be used with a "functional flavour"...

```
//@ model import org.jmlspecs.models.*;

...
/*@
  @   ...
  @   ensures \result
  @           ==> theCollection.equals(\old(theCollection.insert(o)));
  @*/
```

Using Separate Files for Specifications

- Sometimes, it is convenient (or necessary) to separate the JML annotations from the java files
  - in situations where we have no source files (e.g. specifying a library, or in an early specification phase);
  - in order to keep java files "clean" (avoiding cluttering up the code).
- It is recommended that
  - ".spec" or ".jml" for the first case;
  - ".refines-java" for the second (and include a "refines XXX.java" annotation in it).
- When we start by specifying the class, it is recommended to name it "XXX.java-refined" and include, in the implementation "XXX.java" the refine clause "refines XXX.java-refined".