Advanced JML
and more tips and pitfalls

David Cok, Joe Kiniry, and Erik Poll

Eastman Kodak Company, University College Dublin, and Radboud University Nijmegen

Core JML

Remember the core JML keywords were

- `requires`

- `ensures`

- `signals`

- `assignable`

- `normal_behavior`

- `invariant`

- `non_null`

- `pure`

- `\old, \forall, \exists, \result`

More advanced JML features

- **Visibility**

- **Specifying (im)possibility of exceptions**

- **Assignable clauses and datagroups**

- **Aliasing**

- **Specification inheritance, ensuring behavioural subtyping**

- **Specification-only fields: `ghost` and `mode` fields**

- **The semantics of `invariant`**

Visibility

Visibility

JML imposes visibility rules similar to Java, eg.

```
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

```
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.:

```
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.

Exceptions and JML

Exceptional specifications

A method specification can (dis)allow the throwing of exceptions, and specify a exceptional postcondition that should hold in the event an exception is thrown.

There are some implicit rules for (dis)allowing exceptions.

Warning: exceptional specifications are easy to get wrong.

Not allowing any exceptions to be thrown is the simplest approach.

Exceptions allowed by specs

By default, a method is allowed to throw exceptions, but only those listed in its throws clause. So

```
//@ requires 0 <= amount && amount <= balance;
 public int debit(int amount)
                              throws BankException
   { ... }
```

has an implicit clause

```
 signals (BankException) true;
```

and an implicit clause

```
 signals (Exception e) e instanceof BankException;
```

Exceptions allowed by specs

By default, a method is allowed to throw exceptions, *but only those listed in its throws clause.* So

```
//@ requires 0 <= amount && amount <= balance;
 public int debit(int amount)
  { ... }
```

has an implicit clause

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

NB `debit` is now not even allowed to throw an unchecked exception, even though Java does not require a throws clause for these.
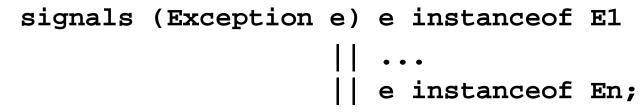
Ruling out exceptions

To forbid a particular exception `SomeException`

1. omit it from throws clause (only possible for unchecked exceptions)

2. add an explicit

    ```
    signals (SomeException) false;
    ```

3. limit the set of allowed exceptions, use a postcondition such as

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

    or, equivalently, us the shorthand for this

    ```
    signals_only E1, ... En;
    ```

Ruling out exceptions

To forbid *all* exceptions

1. **omit all exceptions from throws clause (only possible for unchecked exceptions)**

2. **add an explicit**

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

3. **use keyword normal_behavior to rule out all exceptions**

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

normal_behavior has implicit signals(Exception)false

may vs must throw an exception

Beware of the difference between

    (1) if P holds, then `SomeException` is thrown

and

    (2) if `SomeException` is thrown, then P holds

These are easy to confuse!

(2) can be expressed with `signals`,
(1) can be expressed with `exceptional_behavior`

exceptional_behavior

```
/*@ exceptional_behavior
      requires amount > balance
      signals (BankException e)
              e.getReason.equals("Amount too big")
   @*/
 public int debit(int amount) throws BankException
{ ...  }
```

says a **BankException** *must* be thrown if amount > balance

normal_behavior has implicit 'signals(Exception)false'
exceptional_behavior has implicit 'ensures false'

Example

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

Example

or, equivalently

```
/*@ requires true;
     ensures \old(amount<=balance) && ...
     signals (BankException e)
             \old(amount>balance) && ...
  @*/
 public int debit(int amount) throws BankException
 { ...  }
```

Exceptional behaviour

Moral: to keep things simple, disallow exceptions in specs whenever possible

Eg, for

```
public void arraycopy(int[] src, int destOffset,
                            int[] dest, int destOffset, int leng
      throws NullPointerException,
             ArrayIndexOutOfBoundsException
```

write a spec that disallows any throwing of exceptions, and only worry about specifying exceptional behaviour if this is really needed elsewhere.

Assignable clauses and datagroups

Problems with assignable clauses

Assignable clauses

- **tend to expose implementation details**

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

  ...

  //@ assignable x, ....;
  public void m(...) {....}
  ```

- **tend to become very long**

  ```
  //@ assignable x, y, z[*],  ....;
  ```

- **tend to accumulate**

  ```
  //@ assignable x, f.x, g.y, h.z[*], ....;
  ```

Problems with assignable clauses

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

Solution: 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) { ... }
}
```

time and **alarm** are **model fields**, ie. specification-only fields

Datagroups

Datagroups provide an abstraction mechanism for assignable clauses.

There's a default datagroup `objectState` defined in `Object.java`

It's good practice to declare that all instance fields are in `objectState`

Datagroups can be nested

```
public class Timer{
//@ public model JMLDatagroup time, alarm;//@ in objectSta
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) { ... }
}
```

Assignable and arrays

Another implementation, using an array of 6 digits to represent hrs:mns:secs

```
public class ArrayTimer{
  /*@ spec_public @*/ char[] currentTime;


  //@ invariant currentTime != null;
  //@ invariant currentTime.length == 6;


  //@ assignable currentTime[*];
  public void tick() { ... }


  ...
}
```

Datagroups and arrays

Another implementation, using an array of 6 digits to represent hrs:mns:secs

```
public class ArrayTimer{
  //@ public model JMLDatagroup time;
  char[] currentTime; //@ in time;
  //@ maps currentTime[*] \into time;


  //@ invariant currentTime != null;
  //@ invariant currentTime.length == 6;


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


  ...
}
```

Datagroups and interfaces

Datagroups are convenient in specs for interfaces.

```
public interface TimerInterface{
  //@ model instance public JMLDatagroup time, alarm;


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

Classes implementing this interface are free to choose which fields are in the datagroups.

Keyword `instance` is needed, because fields in interfaces are by default static fields in Java. Interfaces in Java do not have instance fields, but in JML they can have *model* instance fields

The problem with assignable clauses

Despite the abstraction possibilities offered by datagroups, assignable clauses remain a bottlecneck both in program specification and in program verification.

Note that the proof obligation corresponding to an assignable clause is a very complicated one, involving a quantification over all fields not mentioned in the assignable clause

Aliasing (revisited)

Aliasing

Aliasing is the root of all evil, for anyone trying to verify imperative programs.

The `ArrayTimer` class just earlier is another nice example to illustrate this.

ArrayTimer example

Recall implementation using an array of 6 digits to represent hrs:mns:secs

```java
public class ArrayTimer{
  char[] currentTime;
  char[] alarmTime;

  //@ invariant currentTime != null;
  //@ invariant currentTime.length == 6;
  //@ invariant alarmTime != null;
  //@ invariant alarmTime.length == 6;

  public void tick() { ... }

  public void setAlarm(...) { ... }
}
```

ArrayTimer example

Things will go wrong if different instances of `ArrayTimer` share the same `alarmTime` array, ie.

```
o.alarmTime == o'.alarmTime
```

for some `o!=o'`

ESC/Java2 may notice this, produce a correct, but puzzling, warning.

ArrayTimer example

To rule out such aliasing of eg. `alarmTime` fields:

```
public class ArrayTimer{
  char[] currentTime;
  //@ invariant currentTime.owner == this;
  ...

  public ArrayTimer( ...){
    ...;
    currentTime = new char[6];
    //@ set currentTime.owner == this;
    ...
  }
```
(**owner** is a so-called ghost field, more about that later)

ArrayTimer example

Things will go wrong if an instance of `ArrayTimer` aliases its `alarmTime` to its `currentTime`, ie.

```
o.alarmTime == o.currentTime
```

ESC/Java2 may notice this, produce a correct, but puzzling warning.

You should add

```
//@ invariant alarmTime != currentTime;
```

to rule out such aliasing.

Specification-only fields: `ghost` and `model` fields

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.

Ghost fields - example

Suppose the informal spec of

```
class SimpleProtocol {

  public void start() { ... }

  public void stop() { ... }
}
```

says that `stop()` may only be invoked after `start()`, and vice versa.

This can be expressed using a ghost field, to represent the state of the protocol.

Ghost fields - example

```
class SimpleProtocol {
 //@ public ghost boolean started;


 //@ requires  !started;
 //@ assignable started;
 //@ ensures    started;
 public void start() {

    ...

    //@ set started = true; }


 //@ requires   started;
 //@ assignable started;
 //@ ensures   !started;
 public void stop() {

    ...

    //@ set started = false; }
```

Ghost fields - example

Maybe the object has some internal state that that records if protocols is in progress, eg.

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

  ...
  public void start() {
      ...
    st = new ProtocolStack(...);
    ...  }


  public void stop() {
      ...
    st = null;
    ...  }
```

Ghost fields - example

There may be a relation between the ghost field and some other field(s), eg.

```
class SimpleProtocol {
  private ProtocolStack st;
  //@ public ghost boolean started;
  //@ invariant started <==> (st !=null);

  //@ requires  !started;
  //@ assignable started;
  //@ ensures    started;
  public void start() { ...  }

  //@ requires   started;
  //@ assignable started;
  //@ ensures   !started;
  public void stop() { ...  }
```

Ghost fields - example

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

```
class SimpleProtocol {
private /*@ spec_public @*/ ProtocolStack st;


 //@ requires    st==null;
 //@ assignable st;
 //@ ensures     st!=null;
 public void start() { ... }


 //@ requires    st!=null;
 //@ assignable st;
 //@ ensures     st==null;
 public void stop() { ... }
```

but this is ugly and exposes implementation details.

Model fields - example

Solution: use a model field

```
class SimpleProtocol {
private ProtocolStack st;

//@ public model boolean started;
//@ private represents started = (st != null);

//@ requires !started;
//@ assigable started;
//@ ensures   started;
public void start() { ... }

//@ requires  started;
//@ assigable started;
//@ ensures  !started;
public void stop() { ... }
```

Model fields - example

A model field also provided an associated datagroup

```
class SimpleProtocol {
private ProtocolStack st;    //@ in started;


//@ public model boolean started;
//@ private represents started = (st != null);


//@ requires !started;
//@ assigable started;
//@ ensures   started;
public void start() { ... }


//@ requires  started;
//@ assigable started;
//@ ensures  !started;
public void stop() { ... }
```

Model vs ghost fields

Difference between ghost and model may be 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'.

Invariants

Invariants

Invariants – aka class invariants – are a common & very useful notion.

In larger programs, the (only) interesting thing to specify are the invariants.

However, the semantics is trickier than expected!

Invariant is implicitly included in pre- and postconditions of method, and in postcondition of constructors. But there's more ...

In fact, invariants are a hot research topic.

When should an invariant hold?

```
public class A {
   B b;
   int i=2; //@ invariant i >= 0


   //@ ensures \result >=0;
   public /*@ pure @*/ int get(){ return i; }


   public void m(){
      i--;
      ... ; // invariant possibly broken
      i++;
   }
}
```
An invariant can temporarily be broken.

When should an invariant hold?

```
public class A {
  B b;
  int i=2; //@ invariant i >= 0

  //@ ensures \result >=0;
  public /*@ pure @*/ int get(){ return i; }

  public void m(){
    i--;
    b.m(...); // invariant possibly broken
    i++;
  }
}
```

This may lead to problems if invocation of `b.m` involves an invoking on the current object.

When should an invariant hold?

Eg, suppose

```
public class B {

...

  public void m(A a){

   ...

   int j = a.get(); //@ assert i>=0;

   ...

  }
```

The spec of `get()` suggests the assertion will be true.

But if `get()` is invoked when `a`'s invariant is broken, all bets are off.

This is known as the call-back problem.

When should an invariant hold?

Solution to the call-back problem:

> A method invariant should hold in all so-called
> visible states, which are all beginning and end
> states of method invocations

So there's more to invariants than just adding them to pre-
and postconditons.

NB *all* invariants of *all* objects should hold in visible states;
this clearly imposes impossible proof obligations.

ESC/Java2 looks only at the invariants of some objects; this
is a source of unsoundness.

Modular verification techniques for invariants are a
challenge, and still a hot topic of research.

When should an invariant hold?

Sometimes you do want to invoke a method at a program point where the invariant is broken. To do this without ESC/Java2 complaining:

When should an invariant hold?

Sometimes you do want to invoke a method at a program point where the invariant is broken. To do this without ESC/Java2 complaining:

- **A private method can be declared as `helper` method**

  ```
  private /*@ helper @*/ void m() { ... }
  ```

  Invariants do not have to hold when such a helper method is called.

  Effectively, such methods are in-lined in verifications

When should an invariant hold?

Sometimes you do want to invoke a method at a program point where the invariant is broken. To do this without ESC/Java2 complaining:

- **A private method can be declared as `helper` method**

  ```
  private /*@ helper @*/ void m() { ... }
  ```

  Invariants do not have to hold when such a helper method is called.

  Effectively, such methods are in-lined in verifications

- **add**

  ```
  //@ nowarn Invariant
  ```

  in the line where this method call occurs.

  NB this is unsafe, and a possible source of unsoundness!

More problems with invariants

```java
public class SortedLinkedList {
  private int i;
  private LinkedList next;
  //@ invariant i > next.i;
  public /*@ pure @*/ int getValue(){ return i; }
  public /*@ pure @*/ int getNext(){ return next; }
  public /*@ pure @*/ int getValue(){ return i; }
  public void inc() { i++; }
}
```

inc won't break this object's invariant, but may break the invariant of the object who has this object as it's next.

More problems with invariants

The essence of the problem is that the invariant of one object o may depend on the state of another object o′.

When verifying the methods of o′, we have no way of knowing if these may break invariants given in other classes . . .

This is one of the sources of unsoundness of ESC/Java(2), and most approaches to modular verification of OO programs to date.

Only recently workable approaches for modular verification of invariants have been proposed, and the last word has not been said on this.