The Java Modeling language

JML

Erik Poll
Digital Security
Radboud University Nijmegen
JML

- formal specification language for sequential Java by Gary Leavens et. al.
  - to specify behaviour of Java classes & interfaces
  - to record detailed design decisions
  by adding annotations to Java source code in Design-By-Contract style, using eg. pre/postconditions and invariants

- Design goal: meant to be usable by any Java programmer

Lots of info on http://www.jmlspecs.org
to make JML easy to use

• JML annotations added as special Java comments, between /*@ .. @*/ or after //@

• JML specs can be in .java files, or in separate .jml files

• Properties specified using Java syntax, extended with some operators
  \old( ), \result, \forallall, \existsists, \Rightarrow, ..
  and some keywords
  requires, ensures, invariant, ....
JML example

public class ePurse{
    private int balance;
    //@ invariant 0 <= balance && balance < 500;

    //@ requires amount >= 0;
    //@ ensures balance <= \old(balance);

    public debit(int amount) {
        if (amount > balance) {
            throw (new BankException("No way"));
        }
        balance = balance - amount;
    }
}
What can you do with this?

- documentation/specification
  - record detailed design decisions & document assumptions (and hence obligations!)
  - precise, unambiguous documentation
    - parsed & type checked
- use tools for
  - runtime assertion checking
    - eg when testing code
  - compile time (static) analyses
    - up to full formal program verification
LOTS of freedom in specifying

• JML specs can be as strong or weak as you want
  Eg for `debit(int amount)`
  ```
  //@ ensures balance == \old(balance)-amount;
  //@ ensures balance <= \old(balance);
  //@ ensures true;
  ```
  Good bottom-line spec to start: give minimal specs (requires, invariants) necessary to rule out (Runtime)Exceptions

• JML specs can be low(er) level
  ```
  //@ invariant f != null;
  ```
  or high(er) level
  ```
  //@ invariant child.parent == this;
  ```
Rest of this talk

• A bit more JML

• Tools, possibilities, related work, etc
exceptional postconditions: signals

/*@ requires amount >= 0;
@ ensures balance <= \old(balance);
@ signals (BankException) balance == \old(balance);
@*/

public debit(int amount) throws BankException {
    if (amount > balance) {
        throw (new BankException("No way"));
    }
    balance = balance - amount;
}

Often specs (should) concentrate on ruling out exceptional behaviour
ruling out exceptions

/*@ normal_behavior
    @ requires amount >= 0 && amount <= balance;
    @ ensures balance <= \old(balance);
    @*/

public debit(int amount) throws BankException{
    if (amount > balance) {
        throw (new BankException("No way"));
    }
    balance = balance - amount;
}

Or omit “throws BankException”
assert and loop_invariant

...  
/*@ assert (\forall int i; 0<= i && i< a.length;  
    a[i] != null );  
@*/

...  

/*@ loop_invariant 0 <= i && i < a.length & 
(\forall int j; 0<= j & j < i;  
    a[i] != null ); 
    decreasing a.length-i;  
@*/

while (a[i] != null) {...}
non_null

• Lots of invariants and preconditions are about references not being null, eg
  ```java
  int[] a; // invariant a != null;
  ```
• Therefore there is a shorthand
  ```java
  //@ non_null @*/ int[] a;
  ```
• But, as most references are non-null, JML adopted this as default. So only nullable fields, arguments and return types need to be annotated, eg
  ```java
  //@ nullable @*/ int[] b;
  ```
• JML will move to adopting JSR308 Java tags for this
  ```java
  @Nullable int[] b;
  ```
Methods without side-effects that are guaranteed to terminate can be declared as pure

   /*@ pure @*/ int getBalance (){
        return balance;
    };

Pure methods can be used in JML annotations

    //@ requires amount < getBalance();
    public debit (int amount)
assignable (aka modifies)

For non-pure methods, frame properties can be specified using assignable clauses, eg

```java
/*@   requires amount >= 0;
    assignable balance;
    ensures balance == \old(balance) - amount;
@*/

void debit()
```
says debit is only allowed to modify the balance field

- NB this does not follow from the postcondition
- Assignable clauses are needed for modular verification
- Fields can be grouped in Datagroups, so that spec does not have to list concrete fields
resource usage

Syntax for specifying resource usage

/*@
 measured_by len; // max recursion depth
 working_space (len*4); // max heap space used
 duration len*24; // max execution time
 ensures \fresh(result); // freshly allocated
@*/

public List(int len) {...
}
model state

interface Connection{
  //@ model boolean opened; // spec-only field

  //@ ensures !opened;
  public Connection(...);

  //@ requires !opened;
  //@ ensures opened;
  public void open();

  //@ requires opened;
  //@ ensures !opened;
  public void close();
}
pointer trouble

References are the main source of trouble, also in verification
Univeres are a type system to control aliasing

class A {
    //@ invariant invA;
   /*@ rep @*/ C c1, c2;
   /*@ rep @*/ B b;
}

class B {
    //@ invariant invB;
   /*@ rep @*/ D d;
}
tools, related work, ...
tool support: runtime assertion checking

• implemented in JMLrac, with JMLunit extension
• annotations provide the test oracle:
  – any annotation violation is an error, except if it is the initial precondition

• Pros
  – Lots of tests for free
  – Complicated test code for free, eg for
    signals (Exception) balance == old(balance);
  – More precise feedback about root causes
    • eg "Invariant X violated in line 200" after 10 sec instead of "Nullpointer exception in line 600" after 100 sec
tool support: compile time checking

- extended static checking
  automated checking of simple specs
    - ESC/Java(2)

- formal program verification tools
  interactive checking of arbitrarily complex specs
    - KeY, Krakatoa, Freeboogie, JMLDirectVCGen....

There is a trade-off between usability & qualifyability. In practice, each tool support its own subset of JML.
testing vs verification

- verification gives complete coverage
  - all paths, all possible inputs

- if testing fails, you get a counterexample (trace);
if verification fails, you typically don't....

- verification can be done before code is complete

- verification requires many more specs
  - as verification is done on a per method basis
  - incl API specs
related work

- **OCL for UML**
  pro: not tied to a specific programming language
  con: idem
  less expressive, and semantics less clear

- **Spec# for C#**
  by Rustan Leino & co at Microsoft Research

- **SparkAda for Ada**
  by Praxis High Integrity System
  Commercially used
tools and tool status

• For Java 1.4
  – JML2 jmlrac
  – ESC/Java2
  – KeY

• For newer Java versions - under construction
  – OpenJML
    based on openjdk
    front end for runtime checking (and ESC)
program verification state-of-the-art

• JML verification tools can cope with typical Java Card code
  – small API, only 100's loc

• Microsoft hypervisor verification Hyper-V using VCC
  – 60 kloc of C code
some ideas...

• Coping with *concurrency*
  Track *thread-ownership* of objects
  marking objects are thread-local or shared,
  to make guarantees about memory-separation between threads.
  Largely supported by type system

• *Traceability* could maybe be supported by naming
  JML annotations
  ```java
  //@ invariant propertyXyz: .... ;
  ```
questions?
Exercise: JML specification for `arraycopy`

```java
/*@ requires ... ;
  ensures ... ;
  */
static void arraycopy (int[] src, int srcPos,
                      int[] dest, int destPos,
                      int len)

  throws NullPointerException,
       ArrayIndexOutOfBoundsException;
```

Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array.
Exercise: JML specification for arraycopy

/*@ requires src != null && dest != null &&
  0 <= srcPos && srcPos + len < src.length &&
  0 <= destPos && srcPos + len < dest.length;

ensures (\forall int i; 0 <= i && i < len;
   dest[dstPos+i] == src[srcPos+i] ) &&
   (* rest unchanged *)
@*/

static void arraycopy (int[] src, int srcPos,
           int[] dest, int destPos,
           int len);
Exercise: JML specification for arraycopy

/*@ requires src != null && dest != null &&
   0 <= srcPos && srcPos + len < src.length &&
   0 <= destPos && srcPos + len < dest.length;

ensures (\forall int i; 0 <= i && i < len;
   dest[dstPos+i] == \old(src[srcPos+i]) &&
   (* rest unchanged *)
   @*/

static void arraycopy (int[] src, int srcPos,
   int[] dest, int destPos,
   int len);
Exercise: JML specification for arraycopy

/*@ requires ... 

\ensure (\forall \text{int } i; 0 \leq i \&\& i < \text{len}; 
    \text{dest[dstPos+i]} == \text{old(src[srcPos+i])}) \&\& 
    (* \text{rest unchanged} *)

@*/

static void arraycopy (int[] src, int srcPos, int[] dest, int destPos, int len);

We don't have to write \text{old(len)} and \text{old(dest)[old(dstPos)+1]} in the postcondition, because all parameters are implicitly \text{old()} in JML postconditions
Defaults and conjoining specs

• Default pre- and postconditions
  //@ requires true;
  //@ ensures true;
  can be omitted

• //@ requires P
  //@ requires Q
  means the same as
  //@ requires P && Q;
Default signals clause?

//@ requires amount >= 0;
//@ ensures balance <= \old(balance);
public debit(int amount) throws BankException

• Can debit throw a BankException, if precondition holds? YES
• Can debit throw a NullPointerException, if the precondition holds? NO. Unlike Java, JML only allows method to throw unchecked exceptions explicitly mentioned in throws-clauses!
• Methods are always allowed to throw Errors
Default signals clause?

• For a method

    //@ public void m throws E1, ... En { ... }

the default is

    //@ signals (E1) true;
    ...
    //@ signals (En) true;
    //@ signals_only E1, ... En;

• Here

    //@ signals_only E1, ... En;

is shorthand for

   /*@ signals (Exception e)
        typeof(e) <: E1 || ... || typeof(e) <: En;
    @*/
Specifying exceptional behaviour is tricky!

• Beware of the difference between
  1. if $P$ holds then exception $E$ must be thrown
  2. if $P$ holds then exception $E$ may be thrown
  3. if exception of type $E$ is thrown then $P$ will hold (in the poststate)

This is what **signals** specifies

• Most often we just want to rule out exceptions
  - and come up with preconditions and invariants to do this

• Ruling out exceptions also helps with certified analyses for PCC, as it rules out many execution paths
requiring & ruling out exceptions

/*@  requires amount <= balance;
ensures ...;
signals (Exception) false;
also
  requires amount > balance;
  ensures false;
  signals (BankException) ...;
@*/

public debit(int amount) throws BankException
requiring & ruling out exceptions

/*@ normal_behavior
   requires amount <= balance;
   ensures ...;
also
   exceptional_behavior
   requires amount > balance;
   signals (BankException) ...;
@*/

public debit(int amount) throws BankException
requiring & ruling out exceptions

/*@ normal_behavior
   requires amount <= balance;
   ensures ...;
also
   exceptional_behavior
   requires amount > balance;
   signals (BankException) ...;
@*/

public debit(int amount) throws BankException
requiring & ruling out exceptions

or simply

/*@ requires amount <= balance;
   ensures ...;
   @*/
   
   public debit(int amount) // throws BankException

Effectively a normal_behavior, since there is no throws clause

Ruling out exceptions, esp. RuntimeExceptions, as much as possible is the natural thing to do - and a good bottom line specification
Visibility and spec_public

The standard Java visibility modifiers (public, protected, private) can be used on invariants and method specs, eg

```java
//@ private invariant 0 <= balance;
```

Visibility of fields can be loosened using the keyword `spec_public`, eg

```java
public class ePurse{
    private //@ spec_public @*/ int balance;

    //@ ensures balance <= \old(balance);
    public debit(int amount)

    allows private field to be used in (public) spec of debit
```

Of course, this exposes implementation details, which is not nice...
Dealing with undefinedness

• Using Java syntax in JML annotations has a drawback
  - what is the meaning of
    //@ requires !(a[3] < 0);
  if a.length == 2 ?
• How to cope with Java expressions that throw exceptions?
  - runtime assertion checker can report the exception
  - program verifier can treat a[3] as unspecified integer

• Moral: write protective specifications, eg
  //@ requires a.length > 4 && !(a[3] < 0);
Methods without side-effects that are guaranteed to terminate can be declared as `pure`

```java
/*@ pure */ int getBalance (){  
    return balance;  
};
```

Pure methods can be used in JML annotations

```java
//@ requires amount < getBalance();
public debit(int amount)
```
assignable

The default assignable clause is

```plaintext
//@ assignable \everything;
```

*Pure methods* are

```plaintext
//@ assignable \nothing;
```

*Pure constructors* are

```plaintext
//@ assignable this.*;
```
Reasoning in presence of late binding

Late binding (aka dynamic dispatch) introduces a complication in reasoning:

which method specification do we use to reason about
....; x.m(); ....
if we don't know the dynamic type of x?

Solutions:
1. do a case distinction over all possible dynamic types of x,
   • ie. x's static type A and all its subclasses

Obviously not modular!
1. insist on **behavioural subtyping**:
   • use spec for m in class A and require that specs for m
     in subclasses are stronger or identical
Behavioural subtyping & substitutivity

• The aim of behavioural subtyping aims to ensure the principle of substitutivity:
  "substituting a subclass object for a parent object will not cause any surprises"

• Well-typed OO languages already ensure this in a weak form, as soundness of subtyping:
  "substituting a subclass object for a parent object will not result in 'Method not found' errors at runtime"
behavioural subtyping

Two ways to achieve behavioural subtyping

1. For any method spec in a subclass, prove that it is implies the spec for that method in the parent class
   - ie prove that the precondition is weaker!
   and the postcondition is stronger

1. Implicitly conjoin method spec in a subclass with method specs in the parent class
   - called specification inheritance, which is what JML uses
   - this guarantees that resulting precondition is weaker, and the resulting postcondition is stronger
Specification inheritance for method specs

Method specs are inherited in subclasses, and required keyword also warns that this is the case.

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

Effective spec of \texttt{m} in \texttt{Child}:

```java
requires true;
ensures
(i>=0 ==> result>=i)
&&
(i<=0 ==> result<=i);
```
Specification inheritance for invariants

Invariants are inherited in subclasses, eg in

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

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

the invariant for the Child is \textit{invChild} && \textit{invParent}
JML invariants
The semantics of invariants

• Basic idea:
  - Invariants have to hold on method entry and exit
  - but may be broken temporarily during a method

• NB invariants also have to hold if an exception is thrown!

• But there's more to it than that...
The callback problem

class A {
    int i;
    int[] a;
    B b;
    //@ invariant 0<=i & i < a.length;

    void inc() {a[i]++;
    }

    void break() {
        int oldi = i; i = -1;
        b.m(); i = oldi;
    }
}

class B {
    A a;

    void m() {
        a.inc(); // possible callback
    }
}

What if b.m() does a callback on inc of that same A object, while its invariant is broken...

invariant temporarily broken
The semantics of invariants

• An invariant can be temporarily broken during a method, but - because of the possible callbacks - it has to hold when any other method is invoked.

• Worse still, one object could break another object's invariant...

• visible state semantics
  all invariants of all objects have to hold in all visible states, ie. entry and exit points of methods
Problems with invariants

- The visible state semantics is *very restrictive*
  - eg, a constructor cannot call out to other methods before it has established the invariant

  It can be loosened in an ad-hoc manner by declaring methods as *helper* methods
  - *helper* methods don't require or ensure invariants
  - effectively, you can think of them as in-lined

- The more general problem: *how to cope with invariants that involve multiple (or aggregate) objects*
  - still an active research area...
  - one solution is to use some notion of *object ownership*
universes & relevant invariant semantics

Current JML approach to weakening visible state semantics for invariants

• universe type system
  - enforces hierarchical nesting of objects

• relevant invariant semantics
  - invariant of outer objects may be broken when calling methods in inner objects
universes for alias control

```java
class A {
    //@ invariant invA;
    //@ rep @*/ C c1, c2;
    //@ rep @*/ B b;
}

class B {
    //@ invariant invB;
    //@ rep @*/ D d;
}
```

- invariants should only depend on owned state
- an object's invariant may be broken when it invokes methods on sub-objects