JML: some experiences and directions for future work

Erik Poll

University of Nijmegen

Outline of this talk

- **The specification language JML**

  Tool-supported Design-by-Contract for Java

- **How & what to specify ?**

  Can we use not(at)ions from UML/OCL ?

JML

(Java Modeling Language)

JML by Gary Leavens et al.

Formal specification language for Java

- to specify behaviour of Java classes
- to record design/implementation decisions

by adding assertions to Java source code, for

- preconditions
- postconditions
- invariants
- . . .

as in Eiffel (Design-by-Contract), but more expressive.

JML by Gary Leavens et al.

Formal specification language for Java

- to specify behaviour of Java classes
- to record design/implementation decisions

by adding assertions to Java source code, for

- preconditions
- postconditions
- invariants
- ...

as in Eiffel (Design-by-Contract), but more expressive.

Goal: JML should be easy to use for any Java programmer.

JML

To make JML easy to use:

- **JML keeps close to Java syntax & semantics**
- **Properties specified as Java boolean expressions, extended with a few operators, such as** `==>`, `\old`, `\result`, `\forall`, `\exists`.
- **JML assertions added as comments in .java file, between** `/*@ ... @*/`, **or after** `//@`.

JML example

Pre- and post-conditions for methods, eg.

```
/*@ requires amount >= 0;
    ensures  balance == \old(balance)-amount &&
             \result == balance;
  @*/
 public int debit(int amount) {
   ...
 }
```

Here `\old(balance)` refers to the value of `balance` before execution of the method.

JML example

JML specs can be as strong or as weak as you want.

```
/*@ requires amount >= 0;
    ensures  true;
  @*/
 public int debit(int amount) {
   ...
 }
```

This default postcondition “`ensures true`” can be omitted.

Design-by-Contract

Pre- and postconditions define a contract between a class and its clients:

- **Client must ensure precondition and may assume postcondition**
- **Method may assume precondition and must ensure postcondition**

Eg, in the example specs for `debit`, it is the obligation of the client to ensure that `amount` is positive. The `requires` clause makes this explicit.

JML example

Exceptional postconditions, saying when exceptions may be thrown, can be specified with `signals` keyword

```
/*@ requires amount >= 0;
    ensures  true;
    signals (ISOException e)
              amount > balance          &&
              balance == \old(balance) &&
              e.getReason()==AMOUNT_TOO_BIG;
  @*/
 public int debit(int amount) {
  ...
 }
```

JML example

Again, specs can be as strong or weak as you want.

```
/*@ requires amount >= 0;
    ensures  true;
    signals  (ISOException) true;
  @*/
 public int debit(int amount) { ...
```

NB this specifies that an `ISOException` is the *only* exception that can be thrown by `debit`

JML example: invariants

Invariants (aka *class* invariants) are properties that must be maintained by all methods, eg

```
public class  Wallet {
  public static final short MAX_BAL = 1000;
  private short balance;
   /*@ invariant 0 <= balance
                 && balance <= MAX_BAL;
     @*/
  ...
```

JML example: invariants

Invariants (aka *class* invariants) are properties that must be maintained by all methods, eg

```
public class  Wallet {
  public static final short MAX_BAL = 1000;
  private short balance;
   /*@ invariant 0 <= balance
                  && balance <= MAX_BAL;
     @*/
   ...
```

Invariants must *also* be preserved if a method throws an exception!

JML example: invariants

Invariants (aka *class* invariants) are properties that must be maintained by all methods, eg

```
public class  Wallet {
  public static final short MAX_BAL = 1000;
  private short balance;
   /*@ invariant 0 <= balance
                   && balance <= MAX_BAL;
     @*/
  ...
```

Note: this invariant is responsible for precondition and exception in `debit`.

JML example

Restrictions on side-effects of methods can be specified with `assignable` keyword

```
/*@ requires    ...
    assignable  balance;
    ensures     ...
  @*/
 public int debit(int amount) {
```

This is also called a frame property

Invariants & frame properties

JML brings Design-by-Contract and very familiar notions from Hoare logic (eg. pre- & postconditions) to Java.

But beware that traditional Hoare logics seriously neglect

- **frame properties, as expressed by `assignable` clauses**
- **class invariants, and when exactly these should hold, and who is responsible for establishing them**

These notions are tricky because of pointers/references.

Naive approach to invariants

The naive approach to invariants: treat them as syntactic sugar, ie. just add them to pre- and postconditions. Eg.

```
/*@ requires ...  && Invariant;
      ensures ...  && Invariant;
  @*/
 public int debit(int amount) {
```

This is not what you want (clients now have to establish invariant before calling method?!) **and does not scale.**

When can a class safely hide an invariant from its clients?

JML Tools

JML Tools

- **parsing** and **typechecking** and **jmldoc**

JML Tools

- **parsing and typechecking and jmldoc**
- **runtime assertion checking jmlc, jmlunit** [Iowa]
  test for violations of assertions during execution

JML Tools

- **parsing** and **typechecking** and **jmldoc**
- **runtime assertion checking jmlc, jmlunit** [Iowa]
  test for violations of assertions during execution
  Great to use in testing phase!

JML Tools

- **parsing and typechecking and jmldoc**
- **runtime assertion checking jmlc, jmlunit** [Iowa]
  test for violations of assertions during execution
  Great to use in testing phase!
- **verification: prove (at compile-time) that contracts are never violated,**

JML Tools

- **parsing** and **typechecking** and **jmldoc**
- **runtime assertion checking** **jmlc, jmlunit** [Iowa]
  test for violations of assertions during execution
  Great to use in testing phase!
- **verification: prove (at compile-time) that contracts are never violated,**
  ranging from automatic checking of simple properties to interactive verification of complex properties:

JML Tools

- **parsing and typechecking and jmldoc**
- **runtime assertion checking jmlc, jmlunit** [Iowa]
  test for violations of assertions during execution
  Great to use in testing phase!
- **verification: prove (at compile-time) that contracts are never violated,**
  ranging from automatic checking of simple properties to interactive verification of complex properties:
  - **extended static checker ESC/Java(2)** [Compaq]

JML Tools

- **parsing and typechecking and jmldoc**
- **runtime assertion checking jmlc, jmlunit** [Iowa]
  test for violations of assertions during execution
  Great to use in testing phase!
- **verification: prove (at compile-time) that contracts are never violated,**
  ranging from automatic checking of simple properties to interactive verification of complex properties:
  - **extended static checker ESC/Java(2)** [Compaq]
  - **Chase** [Néstor]

JML Tools

- **parsing** **and** **typechecking** **and** **jmldoc**
- **runtime assertion checking** **jmlc, jmlunit** [Iowa]
  test **for violations of assertions during execution**
  Great to use in testing phase!
- **verification**: **prove** **(at compile-time) that contracts are never violated,**
  ranging from automatic checking of simple properties to interactive verification of complex properties:
  - **extended static checker ESC/Java(2)** [Compaq]
  - **Chase** [Néstor]
  - **JACK + B** [Gemplus/INRIA]

JML Tools

- **parsing** and **typechecking** and **jmldoc**
- **runtime assertion checking jmlc, jmlunit** [Iowa]
  test for violations of assertions during execution
  Great to use in testing phase!
- **verification: prove (at compile-time) that contracts are never violated,**
  ranging from automatic checking of simple properties to interactive verification of complex properties:
  - **extended static checker ESC/Java(2)** [Compaq]
  - **Chase** [Néstor]
  - **JACK + B** [Gemplus/INRIA]
  - **Krakatoa + Coq** [Orsay]

JML Tools

- **parsing and typechecking and jmldoc**
- **runtime assertion checking jmlc, jmlunit** [Iowa]
  test for violations of assertions during execution
  Great to use in testing phase!
- **verification: prove (at compile-time) that contracts are never violated,**
  ranging from automatic checking of simple properties to interactive verification of complex properties:
  - **extended static checker ESC/Java(2)** [Compaq]
  - **Chase** [Néstor]
  - **JACK + B** [Gemplus/INRIA]
  - **Krakatoa + Coq** [Orsay]
  - **LOOP + PVS** [Nijmegen]

JML Tools

- **parsing** **and** **typechecking** **and** **jmldoc**
- **runtime assertion checking** **jmlc, jmlunit** [Iowa]
  test **for violations of assertions during execution**
  Great to use in testing phase!
- **verification**: **prove** **(at compile-time) that contracts are never violated,**
  ranging from automatic checking of simple properties to interactive verification of complex properties:
  - **extended static checker ESC/Java(2)** [Compaq]
  - **Chase** [Néstor]
  - **JACK + B** [Gemplus/INRIA]
  - **Krakatoa + Coq** [Orsay]
  - **LOOP + PVS** [Nijmegen]

  Verification reveals any hidden assumptions, and forces these to be specified!

JML Tools

- **parsing** and **typechecking** and **jmldoc**
- **runtime assertion checking jmlc, jmlunit** [Iowa]
  test for violations of assertions during execution
  Great to use in testing phase!
- **verification: prove (at compile-time) that contracts are never violated,**
  ranging from automatic checking of simple properties to interactive verification of complex properties:
  - **extended static checker ESC/Java(2)** [Compaq]
  - **Chase** [Néstor]
  - **JACK + B** [Gemplus/INRIA]
  - **Krakatoa + Coq** [Orsay]
  - **LOOP + PVS** [Nijmegen]

  Verification reveals any hidden assumptions, and forces these to be specified!
- **runtime detection of invariants by Daikon** [MIT]

LOOP tool

A compiler which translates Java code to PVS code, providing

- shallow embedding of sequential Java & JML in PVS
- denotational semantics of Java & JML, but still executable to a degree (useful for debugging & verification!)
- Hoare logic
- wp-calculi

Hoare logic and wp-calculi defined & proved sound inside PVS.

Strong points of JML

Strong points of JML

- **Easy to learn**: syntax & semantics very close to Java

Strong points of JML

- **Easy to learn: syntax & semantics very close to Java**
- **Range of tools support possible**

Strong points of JML

- **Easy to learn: syntax & semantics very close to Java**
- **Range of tools support possible**
- **No need for a formal model:**

  the Java source code *is* the formal model

Strong points of JML

- **Easy to learn: syntax & semantics very close to Java**
- **Range of tools support possible**
- **No need for a formal model:**

  the Java source code *is* the formal model

  Consequently:
  - **use of JML can be introduced gradually**

Strong points of JML

- **Easy to learn: syntax & semantics very close to Java**
- **Range of tools support possible**
- **No need for a formal model:**

  the Java source code *is* the formal model

  Consequently:
  - **use of JML can be introduced gradually**
  - **JML can be used for existing (legacy) code & APIs**

Strong points of JML

- **Easy to learn**: syntax & semantics very close to Java
- **Range of tools support possible**
- **No need for a formal model**:

  the Java source code *is* the formal model

  Consequently:
  - use of JML can be introduced gradually
  - JML can be used for existing (legacy) code & APIs
  - No gap between model and implementation

Strong points of JML

- **Easy to learn: syntax & semantics very close to Java**
- **Range of tools support possible**
- **No need for a formal model:**

  the Java source code *is* the formal model

  Consequently:
  - **use of JML can be introduced gradually**
  - **JML can be used for existing (legacy) code & APIs**
  - **No gap between model and implementation**

  But:
  - **JML does not provide (or impose) any design methodology, as UML, B, VDM, . . . do**

The model-code gap

In most formal models there is a gap between the model and the real implementation.
In annotation-based approaches such as JML there is not.

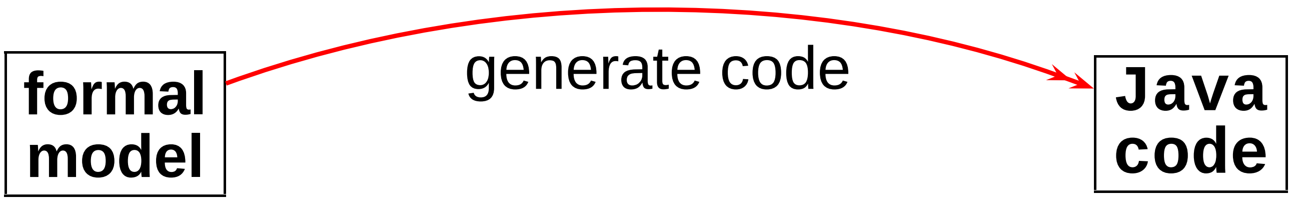
Platonic world of models **`Ugly reality`**

formal model **`Java code`**

The model-code gap

In most formal models there is a gap between the model and the real implementation.
In annotation-based approaches such as JML there is not.

Platonic world of models **`Ugly reality`**



The model-code gap

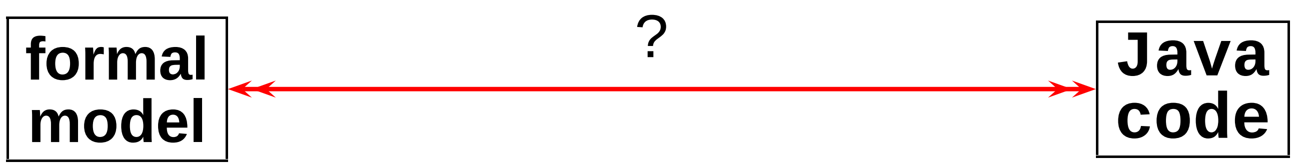
In most formal models there is a gap between the model and the real implementation.
In annotation-based approaches such as JML there is not.

Platonic world of models **`Ugly reality`**

The model-code gap

In most formal models there is a gap between the model and the real implementation.
In annotation-based approaches such as JML there is not.

Platonic world of models **`Ugly reality`**



Con: the gap must be bridged
Pro: nice formal world to work in (eg. $\mathbb{N}$, not **`short`**)

Issues for JML

- **Improving the language and tool support**

  (incl. support for concurrency)

Issues for JML

- **Improving the language and tool support**

  (incl. support for concurrency)

- **How to use JML? What and how to specify?**

  Detailed & complete functional specs as in `debit` example unworkable in practice: it quickly becomes *undoable* and *unreadable*

  Can we use existing (in)formal techniques used for specification here?

What and how to specify?

What and how to specify?

You can of course start with *any* property expressible in JML.

What and how to specify?

You can of course start with *any* property expressible in JML.

But two standard approaches to start:

- **Eliminate runtime exceptions**
- **Specify invariants (for “data consistency”)**

Eliminating runtime exceptions

- **Easy to specify**

  ```
  //@ signals (Exception e) false;
  ```

  and intuitively clear

Eliminating runtime exceptions

- Easy to specify

  `//@ signals (Exception e) false;`

  and intuitively clear

- In practice, trying to rule out runtime exceptions reveals many (hidden) assumptions in code, incl. class invariants

Eliminating runtime exceptions

- **Easy to specify**

  ```
  //@ signals (Exception e) false;
  ```

  and intuitively clear

- **In practice, trying to rule out runtime exceptions reveals many (hidden) assumptions in code, incl. class invariants**

(This approach also proved effective for SPARK/Ada)

Eliminating runtime exceptions

We can push this idea by introducing runtime exceptions for things that shouldn’t happen:

- **Assume checked arithmetic for Java, where numeric overflow results in runtime exception**
- **Assume possible exception for allocation of memory with new**

This departs from the official Java semantics, but this can be justified.

Invariants for data consistency

For non-trival pieces of code, say a file system, writing functional specs quickly become too complicated, and typically you would not even know where to begin.

Invariants for data consistency

For non-trival pieces of code, say a file system, writing functional specs quickly become too complicated, and typically you would not even know where to begin.

(Maybe a nice formal model, seperate from the code, would be useful . . .)

Invariants for data consistency

For non-trival pieces of code, say a file system, writing functional specs quickly become too complicated, and typically you would not even know where to begin.

(Maybe a nice formal model, seperate from the code, would be useful . . .)

However, there are typically many invariants expressing "data consistency" requirements.

Example invariants

```
public class Directory {

 Directory  parent;
  //@ invariant parent==null <==> this==FileSystem.ROOT;

 Directory[]  subdirs;
  /*@ invariant
        (\forall int i; 0<= i && i < subdirs.length
                      ; subdirs[i] != null &&
                        subdirs[i].parent == this);
    @*/
```

Example invariants

```
public class Directory {

 Directory  parent;
  //@ invariant parent==null <==> this==FileSystem.ROOT;

 Directory[]  subdirs;
  /*@ invariant
        (\forall int i; 0<= i && i < subdirs.length
                      ; subdirs[i] != null &&
                        subdirs[i].parent == this);
    @*/
```

Such invariants also typically needed to rule out exceptions...

What and how to specify?

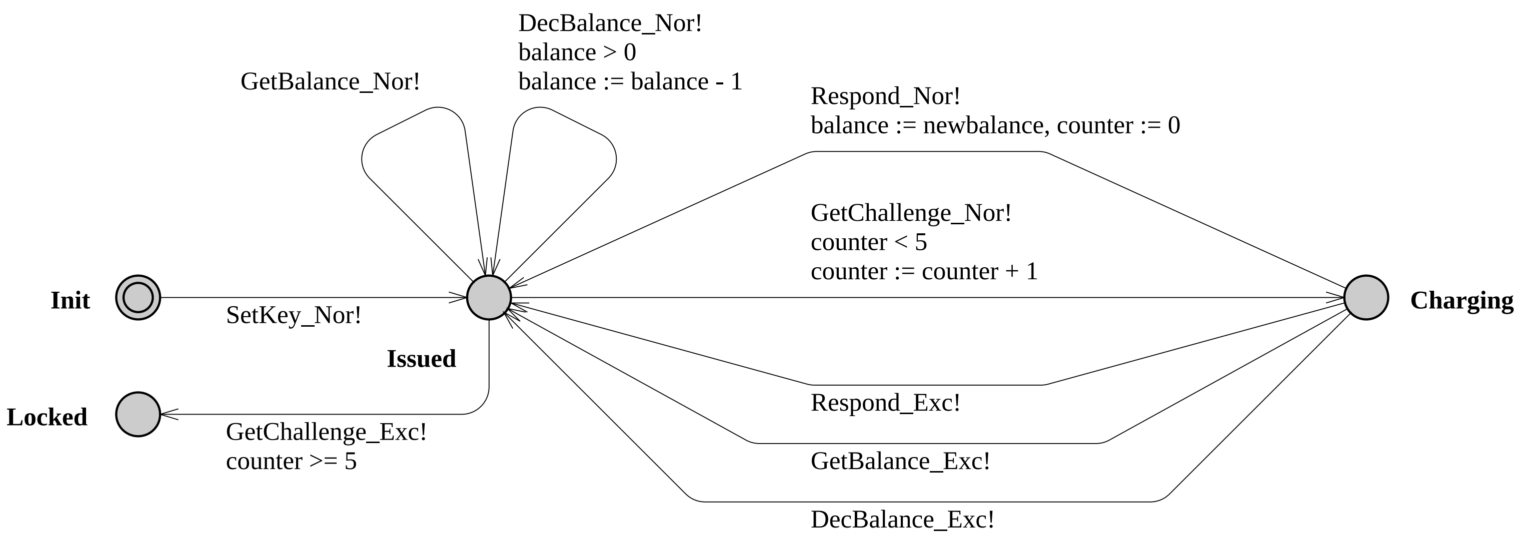
In addition to these 2 standard approaches to write specs, can we re-use existing specification techniques to develop JML specs?

For example

- **State machines/automata aka UML state diagrams**
- **UML class diagrams**
- **OCL constraints**

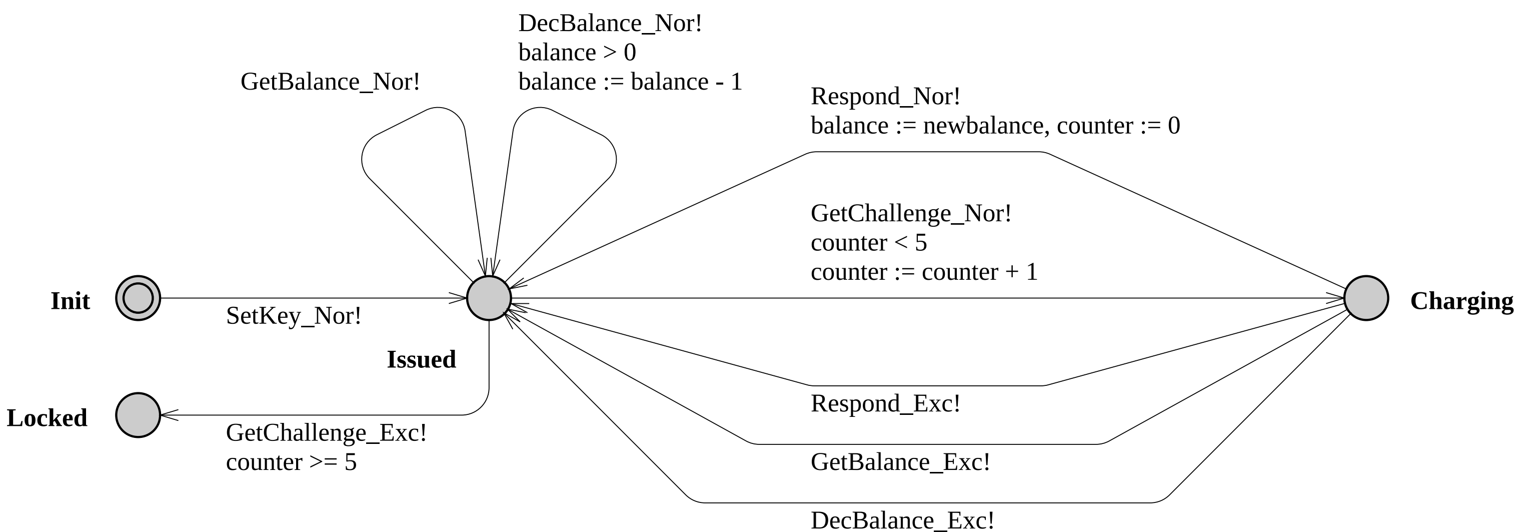
State diagrams

UML state diagrams, (extended) finite state machines, state transition diagrams, . . . are convenient for specification, to specify allowed method invocation traces.



State diagrams

UML state diagrams, (extended) finite state machines, state transition diagrams, . . . are convenient for specification, to specify allowed method invocation traces.



Specifying such properties in JML is possible, but clumsy.

State diagram in JML

```
/*@ invariant
  @  (mode==INIT || mode==ISSUED || mode==CHARGING || mode==LOCKED);
  @*/

/*@ constraint
  @  (\old(mode)==LOCKED ==> mode==LOCKED) &&
  @  (mode==LOCKED ==> \old(mode)==ISSUED || \old(mode)==LOCKED) &&
  @  ...

 /*@  requires    mode==ISSUED || mode==CHARGING;
   @  assignable  mode;
   @  ensures     mode==ISSUED;
   @  signals    (ISOException e) mode==\old(MODE);
   @*/
 private void getValue(APDU apdu) {
```

State diagrams

AutoJML tool (by Martijn Oostdijk and Engelbert Hubbers) translates state diagrams to lots of such JML annotations

Uses auxiliary variable for state (`ghost` or `model` field)

Supports various imput formats, incl. UML, Uppaal, Casper.
Can also be used for *security automata*.

UML vs JML?

Are there other parts of UML that can be useful to produce JML specs?

Experiment in translating (by hand) UML/OCL specs for the BART case study (Bay Area Rapid Transport System) to JML.

UML/OCL model of BART consist of associations in class diagrams and additional OCL contraints

Class diagrams: associations

Associations in class diagram with multiplicity 1-to-1 (eg. between `Station` and `StationPlatform`) **give rise to JML invariants,**

```
public class Station {
   private StationPlatform  pf;
   //@ invariant pf != null;
   //@ invariant pf.getStation() == this;
  ...
```

and something similar in `StationPlatform`.

Class diagrams: associations

Invariants such as

```
//@ invariant pf.getStation() == this;
```

are tricky, as they involve two objects: the invariant will be broken when one of the objects is under construction

Class diagrams: associations

Invariants such as

```
//@ invariant pf.getStation() == this;
```

are tricky, as they involve two objects: the invariant will be broken when one of the objects is under construction

Ad-hoc solution: put invariant only in one of the classes.

A general solution to deal with such situations would be nicer . . .

Class diagrams: associations

Associations using *, eg. 1-to-*, already have to be dealt with in Java, as opposed to JML

Class diagrams: associations

Associations using *, eg. 1-to-*, already have to be dealt with in Java, as opposed to JML

Associations using 0..1, eg. 1-to-0..1, signal that a reference may be `null`, so that usual invariant about `non_null` should be omitted!
(These 0..1 associations can be *Undefined* **in OCL)**

Class diagrams: associations

Associations using *, eg. 1-to-*, already have to be dealt with in Java, as opposed to JML

Associations using 0..1, eg. 1-to-0..1, signal that a reference may be `null`, so that usual invariant about `non_null` should be omitted!
(These 0..1 associations can be *Undefined* **in OCL)**

Any further OCL invariants given can be turned into JML invariants.

Translating OCL to JML

Some conclusions from translating BART case study:

Translating OCL to JML

Some conclusions from translating BART case study:

- **JML is much more verbose, as Java is much more verbose then UML** (Eg. visibility modifiers, get- & set-methods . . .)

Translating OCL to JML

Some conclusions from translating BART case study:

- **JML is much more verbose, as Java is much more verbose then UML** (Eg. visibility modifiers, get- & set-methods . . .)
- **Tricky differences:**
  - **OCL uses =, but JML often should use `equals`**
  - **all Java references can be `null`, only some OCL refences can be *Undefined***

Translating OCL to JML

Some conclusions from translating BART case study:

- **JML is much more verbose, as Java is much more verbose then UML** (Eg. visibility modifiers, get- & set-methods . . .)
- **Tricky differences:**
  - **OCL uses =, but JML often should use `equals`**
  - **all Java references can be `null`, only some OCL refences can be *Undefined***
- **Much of JML specs have to do with basic issues, eg. references not being `null`, that do not show up in OCL**

  In that sense OCL specs complement JML specs for excluding runtime exceptions.

Related work to JML: SPARK/Ada

Initiative similar to JML, but much more mature, and targeting Ada instead of Java.

SPARK is a subset of Ada95, extended with annotations to enable tool-support, for building high-integrity systems

Tool support for data/information flow, testing, and verification.

Succesfully used for Common Criteria evaluations (eg. for MULTOS certification authority)

Establishing exception freedom claimed as important & useful achievement.

More info: `www.sparkada.com`

Conclusions

- JML represents an opportunity to transfer some formal methods to real use - in industry, or in teaching.
- JML as common specification language of benefit to tool developers *and* users.
- Work to be done in improving the language and tool support, *including support for concurrency!*
- but also, work & experience needed on how to use JML, and find out what & how to specify.

More info: `www.jmlspecs.org`