Introduction to JML

a notation for formally specifying Java programs

Erik Poll

University of Nijmegen

Overview of this talk

- What are formal methods anyway?
- the JML specification language
- two tools for JML:
  1. extended static checking with `escjava`,
     (to be used for your applet)
  2. runtime assertion checking with `jmlc/jmlrac`,
     (to be used for your terminal application)

What are formal methods anyway?

Formal methods for civil engineers

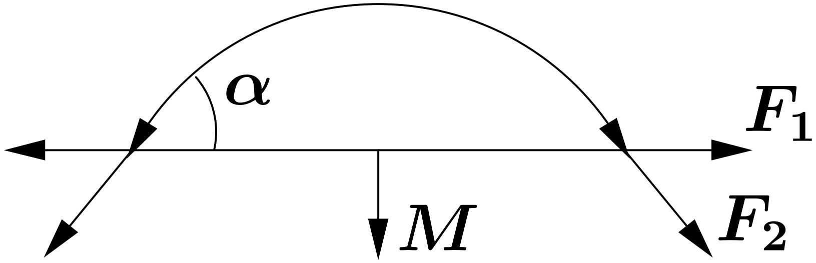
Suppose we build a bridge

How do we know bridge won't collapse ?

Formal methods for civil engineers

reality

(abstract) model



of which properties

$$\frac{M * cos(\alpha) * F_2}{l * \sqrt{h} * \ldots} > M_{max}$$

can be specified and verified

This way we can be certain the bridge won't collapse (modulo modeling errors and abstraction)

Formal methods for software engineers

Suppose we write software for the bridge, to control opening/closing of the bridge, traffic lights, barriers, etc.

```
public class BridgeController{
  public void openBridge()
    {...}
  public void closeBridge()
    {...}
  public void setTrafficlight(Col c)
    {...}
}
```

How can we know that cars will never drive on open bridge?

Formal methods for software engineers

reality

```
public class BridgeController{
 public void openBridge()
 {...}
 public void closeBridge()
 {...}
 public void setTrafficlight(Col c)
 {...}
 }
```

Formal methods for software engineers

reality

```
public class BridgeController{
 public void openBridge()
 {...}
 public void closeBridge()
 {...}
 public void setTrafficlight(Col c)
 {...}
 }
```

Model?
(Do we need one?)
Specifying properties?
Verifying properties?

How can we specify wanted (unwanted) behaviour and ensure that this will always (never) happen?

Formal Methods

Computer scientists have invented a large variety of formal languages to model software and to specify properties about these models, with techniques (logics) to verify these properties.

- ***finite state machines (FSM) aka automata, CSP, process algebra, Z, B, guarded command language, Message Sequence Charts, . . . , Java, . . .***
- ***predicate logic, Hoare logic, temporal logic, . . .***
- ***model checking, theorem proving, . . .***

Formal vs Informal Methods

Why formal as opposed to informal methods ?

(Eg. why not specifications in natural language and reasoning by common-sense?)

- **Precision: formal methods leave no room for ambuity.**
- **Certainty: formal methods can provide more certainty (again, modulo modeling errors and abstraction).**
- **Automation: formal methods can be supported by tools.**

Possible applications of FM

Model the protocol between smartcard and terminal in some security protocol language

1. $terminal \longrightarrow smartcard$: $nonce$
2. $smartcard \longrightarrow terminal$: $\{nonce\}_K$
3. $terminal \longrightarrow smartcard$: ok
4. $smartcard \longrightarrow terminal$: $balance$
5. $terminal \longrightarrow smartcard$: $debitamount$
6. $smartcard \longrightarrow terminal$: $done$

and prove this achieves required security objectives (eg. $terminal$ authenticates $smartcard$) under certain assumptions (eg. only $terminal$ and $smartcard$ know key K).

(Remaining question: does our Java code actually implement the protocol as modeled above?)

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, eg

- preconditions
- postconditions
- class 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, eg

- preconditions
- postconditions
- class 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:

- **Properties are specified as Java boolean expressions, extended with a few operators.**
- **JML assertions are added as comments in .java file, between `/*@` ... `@*/`, or after `//@`.**

Using JML we specify and check properties of *the Java program itself*, not of *some model of our Java program*. Ie. the Java program itself *is* our formal model.

Pre- and postconditions

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.

Pre- and postconditions

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 spec for* `debit`*, it is the obligation of the client to ensure that* `amount` *is positive.

***The* `requires` *clause makes this explicit*.**

Exceptional postconditions

exsures clauses specify when exceptions may be thrown

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

Exceptional postconditions

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

```
/*@ requires amount >= 0;
    ensures  true;
    exsures  (ISOException) true;
  @*/
 public int debit(int amount) throws ISOExceptio
```

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

requires vs. exsures

There is often a trade-off between precondition and exceptional postcondition

```
/*@ requires amount >= 0;
    ensures  true;
    exsures  (ISOException e) true;
  @*/
 public int debit(int amount) throws ISOExceptio
  ...
 }
```

requires vs. exsures

There is often a trade-off between precondition and exceptional postcondition

```
/*@ requires amount >= 0 && amount <= balance;
    ensures  true;
    exsures  (ISOException e) false;
  @*/
 public int debit(int amount) throws ISOExceptio
  ...
 }
```

Maybe "`throws ISOException`*" should now be omitted.*

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

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 exceptio

Example invariants

```
private final Object[] objs;
/*@ invariant
    objs != null
    &&
    objs.length == CURRENT_OBJS_SIZE
    &&
    (\forall int i; 0 <= i && i <= CURRENT_OBJS_SIZE
                  ; objs[i] != null);
  @*/
```

Invariants document design decisions.
Making them explicit helps in understanding the code.

assert clauses

An `assert` clause specifies a property that should hold at some point in the code, eg.

```
private File  getFile ( ... ) {
    try { ...
    } catch (ClassCastException e) { ...
    }
     //@ assert false;
    return null;
  }
```

That’s all, folks!

These examples cover most of what you need to know to start using JML!

There are many more features in JML, but these depend on which tool for JML you use.

Benefits of JML

- **JML specifications provide explicit documentation of contracts and invariants**

Benefits of JML

- **JML specifications provide explicit documentation of contracts and invariants**

  Writing JML specs for code, you make *explicit* assumptions and considerations that have gone into the design of code

Benefits of JML

- JML specifications provide explicit documentation of contracts and invariants

  Writing JML specs for code, you make *explicit* assumptions and considerations that have gone into the design of code

- Such JML specifications make it easier to understand code

Benefits of JML

- **JML specifications provide explicit documentation of contracts and invariants**

  Writing JML specs for code, you make *explicit* assumptions and considerations that have gone into the design of code

- **Such JML specifications make it easier to understand code**

  and should help convincing yourself and others that nothing can go wrong.

Benefits of JML

- JML specifications provide explicit documentation of contracts and invariants

  Writing JML specs for code, you make *explicit* assumptions and considerations that have gone into the design of code

- Such JML specifications make it easier to understand code

  and should help convincing yourself and others that nothing can go wrong.

- Such JML specifications can be used by tools . . .

Tools for JML

- **Runtime assertion checking with `jmlc/jmlrac`.**

  Using `jmlc` and `jmlrac` (instead of `javac` and `java`) performs checks for all JML assertions at runtime:

  any assertion violation results in a special exception.

  To be used for your Java terminal applications

Tools for JML

- **Runtime assertion checking with `jmlc/jmlrac`.**

  Using `jmlc` and `jmlrac` (instead of `javac` and `java`) performs checks for all JML assertions at runtime:

  any assertion violation results in a special exception.

  To be used for your Java terminal applications

- **extended static checking with `escjava`**

  `escjava` proves JML assertions as compile time

  To be used for your Java Card applets

Tools for JML

Runtime assertion checking

- low cost & effort
- easy to do as part of normal testing

Tools for JML

Runtime assertion checking

- **low cost & effort**
- **easy to do as part of normal testing**

Extended checking with ESC/Java

- **higher cost & effort**
- **possible for JavaCard-sized programs**
- **higher assurance: independent of any test suite**
- **checking a spec with ESC/Java *forces* you to specify all the invariants and API contracts that it relies on**

What do we want to specify?

Specification is difficult!

- **Begin by describing the protocol used for every kind of terminal/smartcard interaction in your application (informally). You should be able to relate the state of the terminal/applet to a state in this protocol; the terminal/applet essentially implement a finite state machine.**
- **For all data fields, specify 'sanity' conditions as invariants.**
- **For all methods, specify assumptions it makes on parameters and on fields, as preconditions.**
- **Specifying what you don't want to happen is often easier than specifying what you do want to happen: esp., you don't want any** `Nullpointer`**- or** `ArrayIndexOutOfBoundsExceptions`**.**