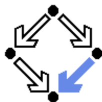
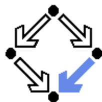
The Java Modeling Language (Part 1)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.uni-linz.ac.at



Overview

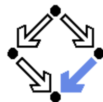


- Since 1999 by Gary T. Leavens et al. (Iowa State University).

  `www.jmlspecs.org`

- A behavioral interface specification language.
  - Syntactic interface and visible behavior of a Java module (interface/class).
  - Tradition of VDM, Eiffel, Larch/C++.
- Fully embedded into the Java language.
  - Java declaration syntax and (extended) expression syntax.
  - Java types, name spaces, privacy levels.
- JML annotations disguised as Java comments.

```
//@ ...
/*@ ...
  @ ... @*/
```

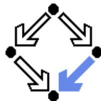


1. **Basic JML**

2. JML Tools

3. More Realistic JML

Basic JML



JML as required for the basic Hoare calculus.

- Assertions.

  `assume`, `assert`.
- Loop assertions.

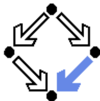
  `loop_invariant`, `decreases`.
- Method contracts.

  `requires`, `ensures`.
- The JML expression language.

  `\forall`, `\exists`, ...

Specifying simple procedural programs.

Assertions



- Definition:

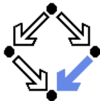
  An assertion is a command that specifies a property which should always hold when execution reaches the assertion.

- JML: two kinds of assertions.
  - `assert` P: P needs verification.
  - `assume` P: P can be assumed.
    - Makes a difference for reasoning tools.
    - A runtime checker must test both kinds of assertions.

```
//@ assert n != 0;
int i = 2*(m/n);
//@ assume i == 2*(m/n);
```

Low-level specifications.

Loop Assertions

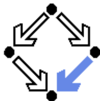


```
int i = n;
int s = 0;
//@ loop_invariant i+s == n;
//@ decreases i;
while (i >= 0)
{
  i = i-1;
  s = s+1;
}
```

- `loop_invariant` specifies a loop invariant, i.e. a property that is true before and after each iteration of the loop.
- `decreases` specifies a termination term, i.e. an integer term that decreases in every iteration but does not become negative.

Useful for reasoning about loops.

Assertions in Methods

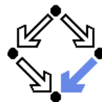


```
static int isqrt(int y)
{
  //@ assume y >= 0;
  int r = (int) Math.sqrt(y);
  //@ assert r >= 0 && r*r <= y && y < (r+1)*(r+1);
  return r;
}
```

- `assume` specifies a condition P on the pre-state.
  - Pre-state: the program state before the method call.
  - The method requires P as the method's precondition.
- `assert` specifies a condition Q on the post-state.
  - Post-state: the program state after the method call.
  - The method ensures Q as the method's postcondition.

Low-level specification of a method.

Design by Contract

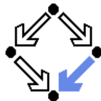


Pre- and post-condition define a contract between a method (i.e. its implementor) and its caller (i.e. the user).

- The method (the implementor) may assume the precondition and must ensure the postcondition.
- The caller (the user) must ensure the precondition and may assume the postcondition.
- Any method documentation must describe this contract (otherwise it is of little use).

The legal use of a method is determined by its contract (not by its implementation)!

Method Contracts

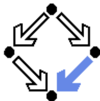


```
/*@ requires y >= 0;
  @ ensures \result >= 0
  @   && \result*\result <= y
  @   && y < (\result+1)*(\result+1); @*/
static int isqrt(int y)
{
  return (int) Math.sqrt(y);
}
```

- `requires` specifies the method precondition
  - May refer to method parameters.
- `ensures` specifies the method postcondition
  - May refer to method parameters and to result value (`\result`).

Higher-level specification of a method.

Postcondition and Pre-State

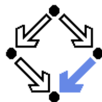


```
// swap a{i] and a[j], leave rest of array unchanged
/*@ requires
  @   a != null &&
  @   0 <= i && i < a.length && 0 <= j && j < a.length;
  @ ensures
  @   a[i] = \old(a[j]) && a[j] == \old(a[i]) &&
  @   (* all a[k] remain unchanged where k != i and k != j *) @*/
static void swap(int[] a, int i, int j)
{ int t = a[i]; a[i] = a[j]; a[j] = t; }
```

- Variable values in postconditions:
  - `x` . . . value of `x` in post-state (after the call).
  - `\old(x)` . . . value of `x` in pre-state (before the call).
  - `\old(E)` . . . expression E evaluated with the value of every variable x in E taken from the pre-state.

Variable values may change by the method call (more on this later).

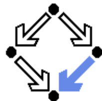
The JML Expression Language



- Atomic Formulas
  - Any Java expression of type `boolean`: `a+b == c`
    - Primitive operators and pure program functions (later).
  - Informal property expression: `(* sum of a and b equals c *)`
    - Does not affect truth value of specification.
- Connectives: `~`P, P`&&`Q, P`||`Q, P`==>`Q, P`<==`Q, P`<==>`Q, P`<=!=>`Q
  - $\neg P, P \wedge Q, P \vee Q, P \Rightarrow Q, Q \Rightarrow P, P \Leftrightarrow Q, \neg(P \Leftrightarrow Q)$.
- Universal quantification: `(\forall` T x; P; Q`)`
  - $\forall x \in T : P \Rightarrow Q$
- Existential quantification: `(\exists` T x; P; Q`)`
  - $\exists x \in T : P \wedge Q$

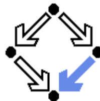
Strongly typed first-order predicate logic with equality.

The JML Expression Language (Contd)



- Sum: `(\sum T x; P; U)`
  - $\sum_{(x \in T) \land P} U$
- Product: `(\product T x; P; U)`
  - $\prod_{(x \in T) \land P} U$
- Minimum: `(\min T x; P; U)`
  - $\min\{U : x \in T \land P\}$
- Maximum: `(\max T x; P; U)`
  - $\max\{U : x \in T \land P\}$
- Number: `(\num_of T x; P; Q)`
  - $\sum_{x \in T : \land P \land Q} 1$
- Set: `new JMLObjectSet {T x | P}`
  - $\{x \in T : P\}$

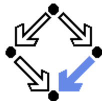
Examples



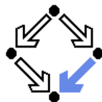
```
// sort array a in ascending order
/*@ requires a != null;
  @ ensures (* a contains the same elements as before the call *)
  @   && (\forall int i; 0 <= i && i < a.length-1; a[i] <= a[i+1]);
  @*/
static void sort(int[] a) { ... }

// return index of first occurrence of x in a, -1 if x is not in a
/*@ requires a != null;
  @ ensures
  @   (\result == -1
  @     && (\forall int i: 0 <= i && i < a.length; a[i] != x)) ||
  @   (0 <= \result && \result < a.length && a[\result] == x
  @     && (\forall int i; 0 <= i && i < \result; a[i] != x));
  @*/
static int findFirst(int[] a, int x) { ... }
```

Examples



```
// swap a{i] and a[j], leave rest of array unchanged
/*@ requires
  @   a != null &&
  @   0 <= i && i < a.length && 0 <= j && j < a.length;
  @ ensures
  @   a[i] = \old(a[j]) && a[j] == \old(a[i]) &&
  @   (\forall k; 0 <= k && k < a.length && k != i && k != j;
  @     a[k] == \old(a[k]));
  @*/
static void swap(int[] a, int i, int j) { ... }
```

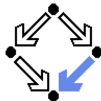


1. Basic JML

2. **JML Tools**

3. More Realistic JML

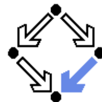
JML Tools



- Type checker `jml`
  - Checks syntactic and type correctness.
- Runtime assertion checker compiler `jmlc`
  - Generates runtime assertions from (some) JML specifications.
- JML skeleton specification generator `jmlspec`
  - Generates JML skeleton files from Java source files.
- Document generator `jmldoc`
  - Generates HTML documentation in the style of `javadoc`.
- Unit testing tool `junit`
  - Generates stubs for the *JUnit* testing environment using specifications as test conditions.

Simple GUI launched by `jml-launcher`.

JML Eclipse Plugin



File Edit Source Refactor Navigate Search Project Run JML Window Help

Setup
Check JML
Clear JML Markers
Enable RAC
Disable RAC
Show RAC
Create Spec File ...
Generate JMLDocs ...
Open Spec File
Show Refinement Sequence

Resource

Navigator
.classpath
.project
C.class
C.java
C.refines-java
Test.class
Test.java

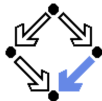
C.refines-java

```
1 /*
2  * Created on 3
3  *
4  * TODO To chan                    ted file go to
5  * Window - Pre                    Code Templates
6  */
7
8 /**
9  * @author schreine
10 *
11 * TODO To change the template for this generated type comment
12 * Window - Preferences - Java - Code Style - Code Templates
13 */
```

Outline
Test
main(String[])
f(int)

Tasks Console
Console

Writable Smart Insert 1 : 1

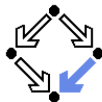


1. Basic JML

2. JML Tools

3. More Realistic JML

More Realistic JML

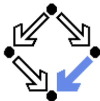


JML for procedural programs with side-effects and errors.

- Side-effects
  - `assignable`, `pure`
- Exceptions
  - `signals`

We also have to deal with the less pleasant aspects of programs.

Side Effects

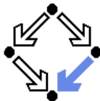


```
static int q, r, x;

/*@ requires b != 0;
  @ assignable q, r;
  @ ensures a == b*q + r && sign(r) == sign(a) &&
  @   (\forall int r0, int q0; a == b*q0+r0 && sign(r) == sign(a);
  @       abs(q) <= abs(q0)) @*/
static void quotRem(int a, int b)
{ q = a/b; r = a%b; }
```

- `assignable` specifies the variables that method may change.
- Default: `assignable \everything`.
  - Method might change any visible variable.
- Possible: `assignable \nothing`.
  - No effect on any variable.

Pure Program Functions



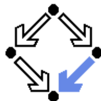
```
static /*@ pure @*/ int sign(int x)
{
  if (x == 0)
    return 0;
  else if (x > 0)
    return 1;
  else
    return -1;
}

static /*@ pure @*/ int abs(int x)
{ if (x >= 0) return x; else return -x; }
```

- Pure program functions may be used in specification expressions.
  - `pure` implies `assignable \nothing`.

JML considers pure program functions as mathematical functions.

Exceptions

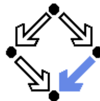


```
static int balance;

/*@ assignable balance;
  @ ensures \old(balance) >= amount
  @   && balance = \old(balance)-amount;
  @ signals(DepositException e) \old(balance) < amount
  @   && balance == \old(balance); @*/
static void withdraw(int amount) throws DepositException
{
  if (balance < amount) throw new DepositException();
  balance = balance-amount;
}
```

- This method has two ways to return.
  - Normal return: the postcondition specified by `ensures` holds.
  - Exceptional return: an exception is raised and the postcondition specified by `signals` holds.

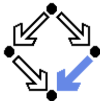
Exceptions



- Default: `signals(Exception e) true;`
  - Instead of a normal return, method may also raise an exception without any guarantee for the post-state.
  - Even if no `throws` clause is present, runtime exceptions may be raised.
- Consider: `signals(Exception e) false;`
  - If method returns by an exception, `false` holds.
  - Thus the method must not raise an exception (also no runtime exception).

We also have to take care to specify the exceptional behavior of a method!

Preconditions versus Exceptions



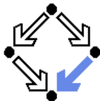
```
/*@ requires (\exists int x; ; a == x*b);
  @ ensures a == \result*b; @*/
static int exactDivide1(int a, int b) { ... }

/*@ ensures (\exists int x; ; a == x*b) && a == \result*b;
  @ signals(DivException e) !(\exists int x; ; a == x*b) @*/
static int exactDivide2(int a, int b) throws DivException { ... }
```

- `exactDivide1` has precondition $P :\Leftrightarrow \exists x : a = x \cdot b$.
  - Method must not be called, if P is false.
  - It is the responsibility of the caller to take care of P.
- `exactDivide2` has precondition true.
  - Method may be also called, if P is false.
  - Method must raise `DivException`, if P is false.
  - It is the responsibility of the method to take care of P.

Different contracts!

Lightweight Specifications



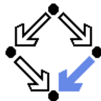
This is the contract format we used up to now.

```
/*@ requires ...;
  @ assignable ...;
  @ ensures ...;
  @ signals ...; @*/
```

- Convenient form for simple specifications.
- If some clauses are omitted, their value is *unspecified*.

So what does a (partially) unspecified contract mean?

Method Underspecification

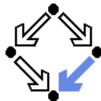


If not specified otherwise, client should assume weakest possible contract:

- `requires false;`
  - Method should not be called at all.
- `assignable \everything;`
  - In its execution, the method may change any visible variable.
- `ensures true;`
  - If the method returns normally, it does not provide any guarantees for the post-state.
- `signals(Exception e) true;`
  - Rather than returning, the method may also throw an arbitrary exception; in this case, there are no guarantees for the post-state.

Defensive programming: for safety, client should avoid implicit assumptions.

Method Underspecification

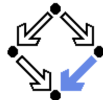


If not specified otherwise, method should implement strongest possible contract:

- `requires true;`
  - Method might be called in any pre-state.
- `assignable \nothing;`
  - In its execution, the method must not change any visible variable.
- `signals(Exception e) false;`
  - Method should not throw any exception.

Defensive programming: for safety, method should satisfy implicit client assumptions (as far as possible).

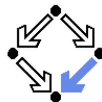
Heavyweight Specifications



```
/*@ public normal_behavior
  @   requires ...;
  @   assignable ...;
  @   ensures ...;
  @ also public exceptional_behavior
  @   requires ...;
  @   assignable ...;
  @   signals(...) ...; @*/
```

- A normal behavior and (one or multiple) exceptional behaviors.
  - Method must implement all behaviors.
- Each behavior has a separate precondition.
  - What must hold, such that method can exhibit this behavior.
  - If multiple hold, method may exhibit any corresponding behavior.
  - If none holds, method must not be called.
- For each behavior, we can specify
  - the visibility level (later), the assignable variables, the postcondition.

Heavyweight Specification Defaults

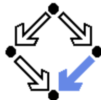


If not specified otherwise, we have the following defaults:

- `requires true;`
  - Method may be called in any state.
- `assignable \everything;`
  - In its execution, the method may change every visible variable.
- `ensures true;`
  - After normal return, no guarantees for the post-state.
- `signals(Exception e) true;`
  - Rather than returning, the method may also throw an arbitrary exception; then there are no guarantees for the post-state.

Method must not make assumptions on the pre-state, caller must not make assumptions on the method behavior and on the post-state.

Example



```
static int balance;

/*@ public normal_behavior
  @   assignable balance;
  @   requires balance >= amount
  @   ensures balance = \old(balance)-amount;
  @ also public exceptional_behavior
  @   requires balance < amount
  @   assignable \nothing;
  @   signals(DepositException e) true;
  @*/
static void withdraw(int amount) throws DepositException
{
  if (balance < amount) throw new DepositException();
  balance = balance-amount;
}
```

Clearer separation of normal behavior and exceptional behavior.