The Java Modeling Language – a Basis for Static and Dynamic Verification

Wolfgang Ahrendt

Chalmers University of Technology, Gothenburg, Sweden

School on Runtime Verification
Praz sur Arly
19 Mar. 2018

Formal Methods: Trace Focus vs. Data Focus

(the following is deliberately simplified)

Runtime Verif.	Static Verif.	Properties	Specifications
Runtime Trace Checking	Model Checking	valid traces (+ some data)	temporal logics, automata, regular languages (+ extensions)
Runtime Assertion Checking	Deductive Verification	valid data in specific code locations (+ some trace info)	first-order assertion languages (+ extensions)

JML (Java Modeling Language)

Outline of Lecture

1. JML – the language
2. Static Verification of JML
3. Runtime Assertion Checking of JML

Literature for this Lecture

KeY Book W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. Schmitt, M. Ulbrich, editors.
Deductive Software Verification - The KeY Book
Vol 10001 of *LNCS*, Springer, 2016
(E-book at `link.springer.com`)

JML Chapter M. Huisman, W. Ahrendt, D. Grahl, M. Hentschel.
Formal Specification with the Java Modeling Language
Chapter 7 in [KeY Book]

Further reading available at
`www.eecs.ucf.edu/~leavens/JML//index.shtml`

Part I

JML – The Language

Unit Specifications

In the object-oriented setting:

Units to be specified are interfaces, classes, and their methods

We start with method specifications.

Method specifications *potentially* refer to:

- initial values of formal parameters
- result value
- prestate and poststate

Specifications as Contracts

To stress the different roles – obligations – responsibilities in a specification:

widely used analogy of the specification as a *contract*

"Design by Contract" methodology (Meyer, 1992, EIFFEL)

Contract between caller and callee (called method)

callee guarantees certain outcome *provided* caller guarantees prerequisites

Running Example: ATM.java

```
public class ATM {

    // fields:
    private BankCard insertedCard = null;
    private int wrongPINCounter = 0;
    private boolean customerAuthenticated = false;

    // methods:
    public void insertCard (BankCard card) { ... }
    public void enterPIN (int pin) { ... }
    public int accountBalance () { ... }
    public int withdraw (int amount) { ... }
    public void ejectCard () { ... }

}
```

Informal Specification

very informal Specification of '`enterPIN (int pin)`':

Enter the PIN that belongs to the currently inserted bank card into the ATM. If a wrong PIN is entered three times in a row, the card is confiscated. After having entered the correct PIN, the customer is regarded as authenticated.

Getting More Precise: Specification as Contract

Contract states what is guaranteed under which conditions.

precondition	card is inserted, user not yet authenticated, pin is *correct*
postcondition	user is authenticated

precondition	card is inserted, user not yet authenticated, pin is *incorrect*, `wrongPINCounter < 2`
postcondition	`wrongPINCounter` has been increased by `1`, user is not authenticated

precondition	card is inserted, user not yet authenticated, pin is *incorrect*, `wrongPINCounter >= 2`
postcondition	card is confiscated user is not authenticated

Meaning of Pre/Postcondition pairs

Definition

A **pre/post-condition** pair for a method `m` is
satisfied by the implementation of `m` if:

> *When* `m` *is called in any state that satisfies the precondition then in any terminating state of* `m` *the postcondition is true.*

1. No guarantees are given when the precondition is not satisfied.
2. Termination may or may not be guaranteed (see below).
3. In case of termination, it may be normal or abrupt (see below).

Formal Specification

Formal Specification

Describe contracts with mathematical rigour

Motivation

- High degree of precision
  - often exhibits omissions/inconsistencies
  - avoid ambiguities
- Automation of program analysis
  - runtime verification
  - static verification
  - test case generation

Java Modeling Language (JML)

JML is a specification language tailored to JAVA.

General JML Philosophy

Integrate

- specification
- implementation

in one single language.

$\Rightarrow$ JML is not external to JAVA

JML
is
JAVA + FO Logic + pre/postconditions, invariants + more...

JML/JAVA integration

JML annotations are attached to JAVA programs
by
writing them directly into the JAVA source code files

Ensures compatibility with standard JAVA compiler:

JML annotations live in special JAVA comments,
ignored by JAVA compiler, recognised by JML tools

JML by Example

from the file ATM.java

```
⋮

/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ...

⋮
```

Everything between /* and */ is invisible for JAVA.

JML by Example

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ...
```

But:

A JAVA comment with '@' as its first character
it is *not* a comment for JML tools.

JML annotations appear in JAVA comments starting with @.

JML by Example

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated; @*/
```

equivalent to:

```
//@ public normal_behavior
//@ requires !customerAuthenticated;
//@ requires pin == insertedCard.correctPIN;
//@ ensures customerAuthenticated;
```

JML by Example

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ...
```

What about the intermediate '@'s?

Within a JML annotation, a '@' is ignored:

- if it is the first (non-white) character in the line
- if it is the last character before '*/'.

⇒ The blue '@'s are not *required*, but it's a convention to use them.

JML by Example

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ...
```

This is a **`public`** specification case:

1. it is accessible from all classes and interfaces
2. it can only mention `public` fields/methods of this class

2. Can be a problem. Solution later in the lecture.

JML by Example

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ...
```

Each keyword ending with **`behavior`** opens a 'specification case'.

`normal_behavior` Specification Case

The method guarantees to *not* throw any exception (on the top level), *if the caller guarantees all preconditions of this specification case*.

JML by Example

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ...
```

This specification case has two preconditions (marked by `requires`)

1. `!customerAuthenticated`
2. `pin == insertedCard.correctPIN`

here:
preconditions are *boolean JAVA expressions*

in general:
preconditions are *boolean JML expressions* (see below)

JML by Example

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
```

specifies only the case where both preconditions are true in prestate

the above is equivalent to:

```
/*@ public normal_behavior
  @ requires (     !customerAuthenticated
  @             && pin == insertedCard.correctPIN );
  @ ensures customerAuthenticated;
  @*/
```

JML by Example

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ...
```

This specification case has one postcondition (marked by **ensures**)

- customerAuthenticated

here:
postcondition is *boolean* Java *expressions*

in general:
postconditions are *boolean JML expressions* (see below)

JML by Example

different specification cases are connected by '**also**'.

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @
  @ also
  @
  @ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin != insertedCard.correctPIN;
  @ requires wrongPINCounter < 2;
  @ ensures wrongPINCounter == \old(wrongPINCounter) + 1;
  @*/
public void enterPIN (int pin) {
    if ( ...
```

JML by Example

```
/*@ <spec-case1> also
  @
  @ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin != insertedCard.correctPIN;
  @ requires wrongPINCounter < 2;
  @ ensures wrongPINCounter == \old(wrongPINCounter) + 1;
  @*/
public void enterPIN (int pin) { ...
```

for the first time, JML expression not a JAVA expression

`\old(E)` means: *E* evaluated in the prestate of `enterPIN`.

E can be any (arbitrarily complex) JML expression.

JML by Example

```
/*@ <spec-case1> also <spec-case2> also
  @
  @ public normal_behavior
  @ requires insertedCard != null;
  @ requires !customerAuthenticated;
  @ requires pin != insertedCard.correctPIN;
  @ requires wrongPINCounter >= 2;
  @ ensures insertedCard == null;
  @ ensures \old(insertedCard).invalid;
  @*/
public void enterPIN (int pin) { ...
```

The postconditions state:

'Given the above preconditions, `enterPIN` guarantees:

`insertedCard == null` and `\old(insertedCard).invalid`'

JML by Example

Question:

could it be

```
@ ensures \old(insertedCard.invalid);
```

instead of

```
@ ensures \old(insertedCard).invalid;
```

??

Specification Cases Complete?

consider spec-case-1:

```
@ public normal_behavior
@ requires !customerAuthenticated;
@ requires pin == insertedCard.correctPIN;
@ ensures customerAuthenticated;
```

what does spec-case-1 *not* tell about poststate?

recall: fields of class `ATM`:

```
insertedCard
customerAuthenticated
wrongPINCounter
```

what happens with `insertCard` and `wrongPINCounter`?

Completing Specification Cases

completing spec-case-1:

```
@ public normal_behavior
@ requires !customerAuthenticated;
@ requires pin == insertedCard.correctPIN;
@ ensures customerAuthenticated;
@ ensures insertedCard == \old(insertedCard);
@ ensures wrongPINCounter == \old(wrongPINCounter);
```

Completing Specification Cases

completing spec-case-2:

```
@ public normal_behavior
@ requires !customerAuthenticated;
@ requires pin != insertedCard.correctPIN;
@ requires wrongPINCounter < 2;
@ ensures wrongPINCounter == \old(wrongPINCounter) + 1;
@ ensures insertedCard == \old(insertedCard);
@ ensures customerAuthenticated
@         == \old(customerAuthenticated);
```

Completing Specification Cases

completing spec-case-3:

```
@ public normal_behavior
@ requires insertedCard != null;
@ requires !customerAuthenticated;
@ requires pin != insertedCard.correctPIN;
@ requires wrongPINCounter >= 2;
@ ensures insertedCard == null;
@ ensures \old(insertedCard).invalid;
@ ensures customerAuthenticated
@         == \old(customerAuthenticated);
@ ensures wrongPINCounter == \old(wrongPINCounter);
```

Assignable Clause

unsatisfactory to add

@ **ensures** *loc* == **\old**(*loc*);

for all locations *loc* which *do not* change

instead:
add assignable clause for all locations which *may* change

@ **assignable** loc_1, . . . , loc_n;

Meaning: No location other than $loc_1, \ldots, loc_n$ can be assigned to.

Special cases:

No location may be changed:

@ **assignable** **\nothing**;

Unrestricted, method allowed to change anything:

@ **assignable** **\everything**;

Specification Cases with Assignable

completing spec-case-1:

```
@ public normal_behavior
@ requires !customerAuthenticated;
@ requires pin == insertedCard.correctPIN;
@ ensures customerAuthenticated;
@ assignable customerAuthenticated;
```

Specification Cases with Assignable

completing spec-case-2:

```
@ public normal_behavior
@ requires !customerAuthenticated;
@ requires pin != insertedCard.correctPIN;
@ requires wrongPINCounter < 2;
@ ensures wrongPINCounter == \old(wrongPINCounter) + 1;
@ assignable wrongPINCounter;
```

Specification Cases with Assignable

completing spec-case-3:

```
@ public normal_behavior
@ requires insertedCard != null;
@ requires !customerAuthenticated;
@ requires pin != insertedCard.correctPIN;
@ requires wrongPINCounter >= 2;
@ ensures insertedCard == null;
@ ensures \old(insertedCard).invalid;
@ assignable insertedCard,
@            insertedCard.invalid,
```

Assignable Groups

You can specify groups of locations as assignable, using '$*$'.

example:

```
@ assignable o.*, a[*];
```

makes all fields of object o and all positions of array a assignable.

JML Modifiers

JML extends the JAVA modifiers by additional modifiers

The most important ones are:

- `spec_public`
- `pure`
- `nullable`
- `non_null`
- `helper`

JML Modifiers: `spec_public`

In `enterPIN` example, pre/postconditions made heavy use of class fields

But: **`public`** specifications can access only **`public`** fields

Not desired: make all fields mentioned in specification **`public`**

Control visibility with `spec_public`

- Keep visibility of JAVA fields **`private/protected`**
- If needed, make them public *only in specification* by **`spec_public`**

```
private /*@ spec_public @*/ BankCard insertedCard = null;
private /*@ spec_public @*/ int wrongPINCounter = 0;
private /*@ spec_public @*/ boolean customerAuthenticated
                                     = false;
```

(Different solution: use specification-only fields; see Sect. 7.7 in [JML Chapter]).

JML Modifiers: Purity

It can be handy to use method calls in JML annotations.

Examples:

`o1.equals(o2)`   `li.contains(elem)`   `li1.max() < li2.min()`

But: specifications must not themselves change the state!

Definition ((Strictly) Pure method)

A method is pure iff it always terminates and has no visible side effects on existing objects.
A method is strictly pure if it is pure and does not create new objects.

JML expressions may contain calls to (strictly) pure methods.

Pure methods are annotated by **`pure`** or **`strictly_pure`** resp.

```
public /*@ pure @*/ int max() { ... }
```

JML Modifiers: Purity Cont'd

- **`pure`** puts obligation on implementor not to cause side effects
- It is possible to formally verify that a method is pure
- **`pure`** implies **`assignable \nothing`**;
  (may create new objects)
- **`assignable \strictly_nothing`**;
  expresses that no new objects are created
- Assignable clauses are local to a specification case
- **`pure`** is global to the method

JML Expressions $\neq$ Java Expressions

boolean JML Expressions (to be completed)

- Each side-effect free **`boolean`** Java expression is a **`boolean`** JML expression
- If `a` and `b` are **`boolean`** JML expressions, and `x` is a variable of type `t`, then the following are also **`boolean`** JML expressions:
  - `!a` ("not a")
  - `a && b` ("a and b")
  - `a || b` ("a or b")
  - `a ==> b` ("a implies b")
  - `a <==> b` ("a is equivalent to b")
  - ...
  - ...
  - ...
  - ...

Beyond `boolean` JAVA expressions

How to express the following?

- An array `arr` only holds values ≤ 2.
- The variable `m` holds the maximum entry of array `arr`.
- All `Account` objects in the array `allAccounts` are stored at the index corresponding to their respective `accountNumber` field.
- All instances of class `BankCard` have different `cardNumbers`.

First-order Logic in JML Expressions

JML **`boolean`** expressions extend JAVA **`boolean`** expressions by:

- implication
- equivalence
- quantification

boolean JML Expressions

boolean JML expressions are defined recursively:

boolean JML Expressions

- each side-effect free **boolean** JAVA expression is a **boolean** JML expression
- if a and b are **boolean** JML expressions, and x is a variable of type t, then the following are also **boolean** JML expressions:
  - `!a` (“not a”)
  - `a && b` (“a and b”)
  - `a || b` (“a or b”)
  - `a ==> b` (“a implies b”)
  - `a <==> b` (“a is equivalent to b”)
  - `(\forall t x; a)` (“for all x of type t, a holds”)
  - `(\exists t x; a)` (“there exists x of type t such that a”)
  - `(\forall t x; a; b)` (“for all x of type t fulfilling a, b holds”)
  - `(\exists t x; a; b)` (“there exists an x of type t fulfilling a, such that b”)

JML Quantifiers

in

`(\forall t x; a; b)`

`(\exists t x; a; b)`

a is called "range predicate"

those forms are redundant:

`(\forall t x; a; b)`
equivalent to
`(\forall t x; a ==> b)`

`(\exists t x; a; b)`
equivalent to
`(\exists t x; a && b)`

Pragmatics of Range Predicates

`(\forall t x; a; b)` and `(\exists t x; a; b)`

widely used

Pragmatics of range predicate:

`a` is used to restrict range of `x` further than `t`

Example: "`arr` is sorted at indexes between 0 and 9":

```
(\forall int i,j; 0<=i && i<j && j<10; arr[i] <= arr[j])
```

Using Quantified JML expressions

How to express:

- An array `arr` only holds values ≤ 2.

```
(\forall int i; 0 <= i && i < arr.length; arr[i] <= 2)
```

Using Quantified JML expressions

How to express:

- The variable `m` holds the maximum entry of array `arr`.

```
(\forall int i; 0 <= i && i < arr.length; m >= arr[i])
```

Is this enough?

```
arr.length > 0 ==>
(\exists int i; 0 <= i && i < arr.length; m == arr[i])
```

Using Quantified JML expressions

How to express:

- All `Account` objects in the array `accountArray` are stored at the index corresponding to their respective `accountNumber` field.

```
(\forall int i; 0 <= i && i < maxAccountNumber;
                accountArray[i].accountNumber == i )
```

Using Quantified JML expressions

How to express:

- All existing instances of class `BankCard` have different `cardNumbers`.

```
(\forall BankCard p1, p2;
         p1 != p2 ==> p1.cardNumber != p2.cardNumber)
```

Example: Specifying `LimitedIntegerSet`

```
public class LimitedIntegerSet {
  public final int limit;
  private int arr[];
  private int size = 0;

  public LimitedIntegerSet(int limit) {
    this.limit = limit;
    this.arr = new int[limit];
  }
  public boolean add(int elem) {/*...*/}

  public void remove(int elem) {/*...*/}

  public boolean contains(int elem) {/*...*/}

  // other methods
}
```

Prerequisites: Adding Specification Modifiers

```
public class LimitedIntegerSet {
  public final int limit;
  private /*@ spec_public @*/ int arr[];
  private /*@ spec_public @*/ int size = 0;

  public LimitedIntegerSet(int limit) {
    this.limit = limit;
    this.arr = new int[limit];
  }
  public boolean add(int elem) {/*...*/}

  public void remove(int elem) {/*...*/}

  public /*@ pure @*/ boolean contains(int elem) {/*...*/}

  // other methods
}
```

Specifying `contains()`

```
public /*@ pure @*/ boolean contains(int elem)  {/*...*/}
```

`contains()` is `pure`: no effect on the state + terminates normally

How to specify result value?

Result Values in Postcondition

In postconditions,
one can use '`\result`' to refer to the return value of the method.

```
/*@ public normal_behavior
  @ ensures \result == (\exists int i;
  @                              0 <= i && i < size;
  @                              arr[i] == elem);
  @*/
public /*@ pure @*/ boolean contains(int elem) {/*...*/}
```

Specifying `add()` (spec-case1) – new element can be added

```
/*@ public normal_behavior
  @ requires size < limit && !contains(elem);
  @ ensures \result == true;
  @ ensures contains(elem);
  @ ensures (\forall int e;
  @                  e != elem;
  @                  contains(e) <==> \old(contains(e)));
  @ ensures size == \old(size) + 1;
  @
  @ also
  @
  @ <spec-case2>
  @*/
public boolean add(int elem) {/*...*/}
```

Specifying add() (spec-case2) – new element cannot be added

```
/*@ public normal_behavior
  @
  @ <spec-case1>
  @
  @ also
  @
  @ public normal_behavior
  @ requires (size == limit) || contains(elem);
  @ ensures \result == false;
  @ ensures (\forall int e;
  @                   contains(e) <==> \old(contains(e)));
  @ ensures size == \old(size);
  @*/
public boolean add(int elem) {/*...*/}
```

Specifying `remove()`

```
/*@ public normal_behavior
  @ ensures !contains(elem);
  @ ensures (\forall int e;
  @                  e != elem;
  @                  contains(e) <==> \old(contains(e)));
  @ ensures \old(contains(elem))
  @         ==> size == \old(size) - 1;
  @ ensures !\old(contains(elem))
  @         ==> size == \old(size);
  @*/
public void remove(int elem) {/*...*/}
```

Specifying Data Constraints

So far:
JML used to specify method specifics.

How to specify constraints on class data?, e.g.:

- consistency of redundant data representations (like indexing)
- restrictions for efficiency (like sortedness)

Data constraints are global: all methods must preserve them

Consider LimitedSorted IntegerSet

```
public class LimitedSortedIntegerSet {
  public final int limit;
  private int arr[];
  private int size = 0;

  public LimitedSortedIntegerSet(int limit) {
    this.limit = limit;
    this.arr = new int[limit];
  }
  public boolean add(int elem) {/*...*/}

  public void remove(int elem) {/*...*/}

  public boolean contains(int elem) {/*...*/}

  // other methods
}
```

Consequence of Sortedness for Implementer

method `contains`

- Can employ binary search (logarithmic complexity)
- Why is that sufficient?
- We assume sortedness in prestate

method `add`

- Search first index with bigger element, insert just before that
- Thereby try to establish sortedness in poststate
- Why is that sufficient?
- We assume sortedness in prestate

method `remove`

- (accordingly)

Specifying Sortedness with JML

Recall class fields:

```
public final int limit;
private int arr[];
private int size = 0;
```

Sortedness as JML expression:

```
(\forall int i; 0 < i && i < size;
                arr[i-1] <= arr[i])
```

(What's the value of this if `size < 2`?)

But where in the specification does the red expression go?

Specifying Sorted `contains()`

Can assume sortedness of prestate

```
/*@ public normal_behavior
  @ requires (\forall int i; 0 < i && i < size;
  @                               arr[i-1] <= arr[i]);
  @ ensures \result == (\exists int i;
  @                                 0 <= i && i < size;
  @                                 arr[i] == elem);
  @*/
public /*@ pure @*/ boolean contains(int elem) {/*...*/}
```

`contains()` is *pure*

⇒ sortedness of poststate trivially ensured

Specifying Sorted `remove()`

Can assume sortedness of prestate

Must ensure sortedness of poststate

```
/*@ public normal_behavior
  @ requires (\forall int i; 0 < i && i < size;
  @                               arr[i-1] <= arr[i]);
  @ ensures !contains(elem);
  @ ensures (\forall int e;
  @                 e != elem;
  @                 contains(e) <==> \old(contains(e)));
  @ ensures \old(contains(elem))
  @         ==> size == \old(size) - 1;
  @ ensures !\old(contains(elem))
  @         ==> size == \old(size);
  @ ensures (\forall int i; 0 < i && i < size;
  @                              arr[i-1] <= arr[i]);
  @*/
public void remove(int elem) {/*...*/}
```

Specifying Sorted `add()` (spec-case1) – can add

```
/*@ public normal_behavior
  @ requires (\forall int i; 0 < i && i < size;
  @                              arr[i-1] <= arr[i]);
  @ requires size < limit && !contains(elem);
  @ ensures \result == true;
  @ ensures contains(elem);
  @ ensures (\forall int e;
  @                 e != elem;
  @                 contains(e) <==> \old(contains(e)));
  @ ensures size == \old(size) + 1;
  @ ensures (\forall int i; 0 < i && i < size;
  @                             arr[i-1] <= arr[i]);
  @
  @ also <spec-case2>
  @*/
public boolean add(int elem) {/*...*/}
```

Specifying Sorted add() (spec-case2) – cannot add

```
/*@ public normal_behavior
  @
  @ <spec-case1> also
  @
  @ public normal_behavior
  @ requires (\forall int i; 0 < i && i < size;
  @                           arr[i-1] <= arr[i]);
  @ requires (size == limit) || contains(elem);
  @ ensures \result == false;
  @ ensures (\forall int e;
  @                   contains(e) <==> \old(contains(e)));
  @ ensures size == \old(size);
  @ ensures (\forall int i; 0 < i && i < size;
  @                          arr[i-1] <= arr[i]);
  @*/
public boolean add(int elem) {/*...*/}
```

Factor out Sortedness

So far: 'sortedness' has swamped our specification

We can do better, using

> **JML Class Invariant**
>
> construct for specifying data constraints centrally

1. delete blue and red parts from previous slides
2. add 'sortedness' as JML class invariant instead

JML Class Invariant

```
public class LimitedSortedIntegerSet {

  public final int limit;

  /*@ private invariant (\forall int i;
    @                                   0 < i && i < size;
    @                                   arr[i-1] <= arr[i]);
    @*/

  private /*@ spec_public @*/ int arr[];
  private /*@ spec_public @*/ int size = 0;

  // constructor and methods,
  // without sortedness in pre/postconditions
}
```

JML Class Invariant

- JML class invariant can be placed anywhere in class
- (Contrast: method contract must be in front of its method)
- Custom to place class invariant in front of fields it talks about

Static JML Invariant Example

```
public class BankCard {

  /*@ public static invariant
    @  (\forall BankCard p1, p2;
    @     p1 != p2 ==> p1.cardNumber != p2.cardNumber)
    @*/

  private /*@ spec_public @*/ int cardNumber;

  // rest of class follows

}
```

Class Invariants: Intuition, Notions & Scope

Class invariants must be

- established by
  - constructors (instance invariants)
  - static initialisation (static invariants)
- preserved by all (non-helper) methods
  - assumed in prestate (implicit preconditions)
  - ensured in poststate (implicit postconditions)
  - can be violated during method execution

Scope of invariant

- not limited to it's class/interface
- depends on visibility (`private` vs. `public`) of local state

⇒ An invariant must not be violated by any code in any class

The JML modifier: `helper`

JML helper methods

```
T /*@ helper @*/ m(T p1, ..., T pn)
```

Neither assumes nor ensures any invariant by default.

Pragmatics & Usage of helper methods

- Helper methods are usually **`private`**.
- Used for structuring implementation of public methods (e.g. factoring out reoccurring steps)
- Used in constructors (where invariants have not yet been established)

Additional purpose in KeY context

Normal form, used when translating JML to Dynamic Logic.
(see below)

Referring to Invariants

Aim: refer to invariants of arbitrary objects in JML expressions.

- **`\invariant_for`**`(o)` is a `boolean` JML expression
- **`\invariant_for`**`(o)` is `true` in a state where all invariants of `o` are `true`, otherwise `false`

Pragmatics:

- Use **`\invariant_for(this)`** when local invariant is intended but *not* implicitly given, e.g., in specification of **`helper`** methods.
- Put **`\invariant_for(o)`**, where **`o`** $\neq$ `this`, into

  `requires`/**`ensures`**/**`invariant`**

  to

  assume/guarantee/maintain

  invariant of **`o`** in local implementation

Example of Referring to local Invariant

```
public class Database {
  ...
  /*@ public normal_behavior
    @ requires ...;
    @ ensures  ...;
    @*/
  public void add (Set newItems) {
    ... <rough adding at first> ...;
    cleanUp();
  }
  ...
  /*@ private normal_behavior
    @ ensures \invariant_for(this);
    @*/
  private /*@ helper @*/ void cleanUp() { ... }
  ...
```

Example of Referring to non-local Invariant

Example

If all (non-helper) methods of ATM shall maintain invariant of object stored in insertedCard:

```
public class ATM {
  ...
/*@ private invariant
  @ insertedCard != null ==> \invariant_for(insertedCard);
  @*/
  private BankCard insertedCard;
  ...
```

Example of Referring to non-local Invariant

Alternatively more fine grained:

Example

If method `withdraw` of `ATM` relies on invariant of `insertedCard`:

```
public class ATM {
  ...
  private BankCard insertedCard;
  ...
  /*@ public normal_behavior
    @ requires \invariant_for(insertedCard);
    @ requires <other preconditions>;
    @ ensures <postcondition>;
    @*/
  public int withdraw (int amount) { ... }
  ...
```

Notes on `\invariant_for`

- For non-helper methods, **`\invariant_for(this)`** *implicitly* added to pre- and postconditions!
- **`\invariant_for`**`(expr)` returns true iff `expr` satisfies the invariant of its static type:
  - Given `class B extends A`
  - After executing initialiser `A o = new B();`
    `\invariant_for(o)` is true when `o` satisfies invariants of `A` ,
    `\invariant_for((B)o)` is true when `o` satisfies invariants of `B`.

Recall Specification of `enterPIN()`

```
private /*@ spec_public @*/ BankCard insertedCard = null;
private /*@ spec_public @*/ int wrongPINCounter = 0;
private /*@ spec_public @*/ boolean customerAuthenticated
                                    = false;

/*@ <spec-case1> also <spec-case2> also <spec-case3>
  @*/
public void enterPIN (int pin) { ...
```

last lecture:
all 3 *spec-cases* were **normal_behavior**

Specifying Exceptional Behavior of Methods

`normal_behavior` specification case, with preconditions P,
forbids method to throw exceptions if prestate satisfies P

`exceptional_behavior` specification case, with preconditions P,
requires method to throw exceptions if prestate satisfies P

Keyword `signals` specifies *poststate*, depending on thrown exception

Keyword `signals_only` limits types of thrown exception

Completing Specification of enterPIN()

```
/*@ <spec-case1> also <spec-case2> also <spec-case3> also
  @
  @ public exceptional_behavior
  @ requires insertedCard==null;
  @ signals_only ATMException;
  @ signals (ATMException) !customerAuthenticated;
  @*/
public void enterPIN (int pin) { ...
```

In case `insertedCard` is `null` in prestate:

- enterPIN *must* throw an exception ('**exceptional_behavior**')
- it can only be an ATMException ('**signals_only**')
- method must then ensure `!customerAuthenticated` in poststate ('**signals**')

signals_only Clause: General Case

An exceptional specification case can have one clause of the form

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

where $E_1,\ldots,E_n$ are exception types

Meaning:

If an exception is thrown, it is of type E_1 or … or E_n

signals Clause: General Case

An exceptional specification case can have several clauses of the form

```
signals (E) b;
```

where `E` is exception type, `b` is boolean expression

Meaning:

If an exception of type `E` is thrown, `b` holds afterwards

Allowing Non-Termination

By default, both:

- **`normal_behavior`**
- **`exceptional_behavior`**

specification cases enforce termination

In each specification case, non-termination can be permitted via the clause

`diverges true;`

Meaning:

Given the precondition of the specification case holds in prestate, the method may or may not terminate

Further Modifiers: `non_null` and `nullable`

JML extends the Java modifiers by further modifiers:

- class fields
- method parameters
- method return types

can be declared as

- **`nullable`**: may or may not be **`null`**
- **`non_null`**: must not be **`null`**

non_null: Examples

```
private /*@ spec_public non_null @*/ String name;
```

Implicit invariant '`public invariant name != null;`' added to class

```
public void insertCard(/*@ non_null @*/ BankCard card) {..
```

Implicit precondition '`requires card != null;`' added to each specification case of `insertCard`

```
public /*@ non_null @*/ String toString()
```

Implicit postcondition '`ensures \result != null;`' added to each specification case of `toString`

non_null Default

non_null is default in JML!

⇒ same effect even without explicit '**non_null**'s

```
private /*@ spec_public @*/ String name;
```

Implicit invariant '**public invariant** name != **null**;'
added to class

```
public void insertCard(BankCard card) {..
```

Implicit precondition '**requires** card != **null**;'
added to each specification case of `insertCard`

```
public String toString()
```

Implicit postcondition '**ensures** \result != **null**;'
added to each specification case of `toString`

nullable: Examples

To prevent such pre/postconditions and invariants: '`nullable`'

```
private /*@ spec_public nullable @*/ String name;
```

No implicit invariant added

```
public void insertCard(/*@ nullable @*/ BankCard card) {..
```

No implicit precondition added

```
public /*@ nullable @*/ String toString()
```

No implicit postcondition added to specification cases of `toString`

LinkedList: non_null or nullable?

```
public class LinkedList {
    private Object elem;
    private LinkedList next;
    ....
```

In JML this means:

- All elements in the list are **non_null**
- The list is cyclic, or infinite!

LinkedList: non_null or nullable?

Repair:

```
public class LinkedList {
    private Object elem;
    private /*@ nullable @*/ LinkedList next;
    ....
```

⇒ Now, the list is allowed to end somewhere!

General Behaviour Specification Case

Meaning of a behavior specification case in JML

An implementation of a method *m* satisfying its behavior spec. case must ensure: If property *P* holds in the method's prestate, then one of the following must hold

```
behavior
 requires P;
 diverges D;
 assignable A;
 ensures Q;
 signals_only
          E1,...,Eo;
 signals (E e) S;
```

- *D* holds in the prestate and method *m* does not terminate (default: D=`false`)
- ...

General Behaviour Specification Case

Meaning of a behavior specification case in JML

An implementation of a method *m* satisfying its behavior spec. case must ensure: If property *P* holds in the method's prestate, then one of the following must hold

```
behavior
 requires P;
 diverges D;
 assignable A;
 ensures Q;
 signals_only
         E1,...,Eo;
 signals (E e) S;
```

- ...
- in the reached (normal or abrupt) poststate: All of the following items must hold
  - only heap locations (static/instance fields, array elements) that did not exist in the prestate or are listed in *A* (assignable) may have been changed

General Behaviour Specification Case

Meaning of a behavior specification case in JML

An implementation of a method *m* satisfying its behavior spec. case must ensure: If property *P* holds in the method's prestate, then one of the following must hold

```
behavior
 requires P;
 diverges D;
 assignable A;
 ensures Q;
 signals_only
          E1,...,Eo;
 signals (E e) S;
```

- ...
- in the reached (normal or abrupt) poststate: All of the following items must hold
  - only heap locations ...
  - if *m* terminates normally, then in its poststate property *Q* holds (default: *Q*=`true`)
  - if *m* terminates normally then ...
  - if *m* terminates abruptly then
    - with an exception listed in **`signals_only`** (default: all exceptions of *m*'s throws declaration + `RuntimeException` and `Error`) and
    - for matching **`signals`** clause, the exceptional postcondition *S* holds

General Behaviour Specification Case

Meaning of a behavior specification case in JML

An implementation of a method *m* satisfying its behavior spec. case must ensure: If property *P* holds in the method's prestate, then one of the following must hold

```
behavior
  requires P;
  diverges D;
  assignable A;
  ensures Q;
  signals_only
          E1,...,Eo;
  signals (E e) S;
```

- ...
- in the reached (normal or abrupt) poststate: All of the following items must hold
  - ...
  - **`\invariant_for(this)`** must be maintained (in normal or abrupt termination) by non-`helper` methods

Desugaring: Normal Behavior and Exceptional Behavior

Both **`normal_behavior`** and **`exceptional_behavior`** cases are expressible as general **`behavior`** cases:

Normal Behavior Case

- desugars to '`signals (Throwable e) false;`'

Exceptional Behavior Case

- desugars to '`ensures false`'

Both default to '`diverge false`', but allow it to be overwritten.

Ghost Variables

- Specification-only variables
- Preceded by keyword **`ghost`**
- Updated by **`set`** annotation
- Can be local variables or fields

Typical usage:

- Mimicking state machine specifications in JML
- Storing additional information about program (memory usage, execution time, ...)

Local Ghost Variable Example: Count Loop Iterations

```
public void doLoop(int x) {
  int y=0;
  //@ ghost int z;
  //@ set z = 0;
  while (x > 0) {
    x = x - 1;
    y = y + 2;
    //@ set z = z + 1;
  }
}
```

Ghost Field Example: Resource Usage

```
//@ public static ghost int MEM;
//@ public static final ghost int MAX = ...;

//@ requires MEM + A.size <= MAX;
//@ ensures MEM <= MAX;
public void m() {
        A a = new A();
        //@ set MEM = MEM + A.size;
}
```

Part II

Static Verification of JML: KeY

KeY

KeY is an approach and tool for the

- Formal specification
- Deductive verification

of

- OO software

KeY Project Partners

Karlsruhe Institute of Technology

Bernhard Beckert, Peter H. Schmitt, Mattias Ulbrich

Technical University Darmstadt

Reiner Hähnle, Richard Bubel

Chalmers University

Wolfgang Ahrendt

incl. post-docs and PhD students

KeY in 30 seconds

- Dynamic logic as program logic
- Verification = symbolic execution + induction/invariants
- Sequent calculus
- Prover is interactive + automated
- most elaborate KeY instance KeY-Java
  - Java as target language
  - Supports specification language JML

JML to Dynamic Logic

Major components of KeY-Java

- Proof Obligation Generator
  - input: Java files containing JML specs
  - output: proof obligations in Dynamic Logic (DL) for Java
- KeY Prover
  - executing a sequent calculus for DL
- KeYTestGen
  - verification based test generation

From JML via Normalised JML to Proof Obligations (PO)

```
public class A {
 /*@ public normal_behavior
   @ requires <Precondition>;
   @ ensures <Postcondition>;
   @ assignable <locations>;
   @*/
 public int m(params) {..}
}
```

Normalisation → Normlaised JML

PO Generation ↙

Proof obligation as DL formula

$$pre \rightarrow \langle \texttt{m(params);} \rangle (post \ \wedge \ frame)$$

Normalisation by Example

```
/*@ public normal_behavior
  @ requires c.id >= 0;
  @ ensures \result == ( ... );
  @*/
  public boolean addCategory(Category c) {
```

becomes

```
/*@ public behavior
  @ requires c.id >= 0;
  @ ensures \result == ( ... );
  @ signals (Throwable exc) false;
  @*/
  public boolean addCategory(Category c) {
```

Normalisation by Example

```
/*@ public behavior
  @ requires c.id >= 0;
  @ ensures \result == ( ... );
  @ signals (Throwable exc) false;
  @*/
  public boolean addCategory(Category c) {
```

becomes

```
/*@ public behavior
  @ requires c.id >= 0;
  @ requires c != null;
  @ ensures \result == (...);
  @ signals (Throwable exc) false;
  @*/
  public boolean addCategory(/*@ nullable @*/ Category c) {
```

Normalisation by Example

```
/*@ public behavior
  @ requires c.id >= 0;
  @ requires c != null;
  @ ensures \result == (...);
  @ signals (Throwable exc) false;
  @*/
  public boolean addCategory(/*@ nullable @*/ Category c) {
```

becomes

```
/*@ public behavior
  @ requires c.id >= 0;
  @ requires c != null;
  @ requires \invariant_for(this);
  @ ensures \result == (...);
  @ ensures \invariant_for(this);
  @ signals (Throwable exc) false;
  @ signals (Throwable exc) \invariant_for(this);
  @*/
public /*@ helper @*/
 boolean addCategory(/*@ nullable @*/Category c) {
```

Generating DL-PO from (normalised) JML

Postcondition *post* states either

- that no exception is thrown or
- that in case of an exception the exceptional postcondition holds

How to refer to an exception in post-state?

$$pre \rightarrow \{\mathtt{heapAtPre} := \mathtt{heap}\} \left\langle \begin{array}{l} \texttt{exc = }\textbf{null}\texttt{;} \\ \textbf{try}\texttt{ \{} \\ \quad \texttt{result = m(args);} \\ \texttt{\} }\textbf{catch}\texttt{ (Throwable e)\{exc = e;\}} \end{array} \right\rangle (post \ \wedge \ frame)$$

Proof Guiding Annotations: Loop Invariants

```
public int[] a;
/*@ public normal_behavior
  @  ensures (\forall int x; 0<=x && x<a.length; a[x]==1);
  @*/
public void m() {
  int i = 0;
  /*@ loop_invariant
    @  0 <= i && i <= a.length &&
    @  (\forall int x; 0<=x && x<i; a[x]==1);
    @ assignable a[*];
    @*/
  while(i < a.length) {
    a[i] = 1;
    i++;
  }
}
```

Part III

Runtime Assertion Checking of JML: OpenJML

OpenJML

OpenJML: tool for JML based Java development

Features:

- runtime assertion checking (RAC)
- *lightweight* static verification

Main developer: David Cok

Main Concerns in Runtime Assertion Checking

OpenJML has to address:

- Operationalisation of quantifiers
- **`\old`**(...) expressions:
  evaluation *after* execution, but in *before*-state
- Specification-only expressions (ghost/model variables)
- Methods with multiple exit points
- Exceptional postconditions
- Undefinedness ($x/0$)
- ...

Requirements on Runtime Assertion Checkers

Transparency If no assertions is violated,
RAC does not change *functional* behaviour

Isolation Annotation violations report “where” they occur

Trustworthy Only real violations are reported

Desired:

- Minimise runtime overhead
  But RAC tools (incl. OpenJML) are not good in that

Typical RAC Usage

- Special compilation option
- Code instrumentation (on bytecode level):
  Inserts checks at appropriate points
- Execution with run-time checks enabled during debugging phase
- Final version: run-time checks disabled

Example Output

```
CStudent.java:67: JML postcondition is false
    public void activityBonus(int bonus) {
                ^

CStudent.java:58: Associated declaration: CStudent.java:67:
          ensures getCredits() ==
          ^

ExecuteCStudent.java:9: JML postcondition is false
        s.activityBonus(5);
                       ^
```

Part IV

Wrap Up and Perspectives

Java Modeling Language

- Specification language
- Data properties "at" specific code positions
- Combines Java (oo) concepts with first-order logic

Used for:

- Static verification (KeY)
- Runtime assertion checking (OpenJML)
- Combined Static and Runtime Verification (StaRVOOrS)
- Test Case Generation (JMLUnitNG, KeYTestGen)

Credits

Slides on

- Runtime Assertion Checking (Part III)
- Ghost Variables

based on material by Marieke Huisman

JML Chapter M. Huisman, W. Ahrendt, D. Grahl, M. Hentschel.
Formal Specification with the Java Modeling Language
Chapter 7 in [KeY Book]