Formal Specification and Verification of Object-Oriented Programs

The Java Modeling Language: Basic Language Features (Part II)

TECHNISCHE UNIVERSITÄT DARMSTADT

Recapture of Previous Lecture

Focus: Basic JML expressions and normal behavior specification of methods

```
/*@ < visibility modifier > normal_behavior
  @ requires P;
  @ ensures Q;
  @ assignable < set of locations >;
  @*/
T m(T1 p1, ..., Tn pn)
```

- JML method specification attached to methods as comments (must be placed directly before the method declaration)
- visibility of a specification case determines access: only accessible fields and methods can be used
  - **spec_public** can be used to lift visibility of fields for specifications (similar: **spec_protected**)
- **normal_behavior** opens a normal behavior specification case

Recapture of Previous Lecture

TECHNISCHE UNIVERSITÄT DARMSTADT

Focus: Basic JML expressions and normal behavior specification of methods

```
/*@ <visibility modifier> normal_behavior
  @ requires P;
  @ ensures Q;
  @ assignable <set of locations>;
  @*/
T m(T1 p1, ..., Tn pn)
```

- keyword
  - **requires** means that boolean JML expression P denotes a precondition
  - **ensures** means that boolean JML expression Q denotes a postcondition

  of method `m`
- several requires clauses with preconditions $P_1, \ldots, P_n$ are equivalent to a single requires clause with precondition P_1 && ... && P_n (same for ensures)

Recapture of Previous Lecture

Focus: Basic JML expressions and normal behavior specification of methods

```
/*@ <visibility modifier> normal_behavior
  @ requires P;
  @ ensures Q;
  @ assignable <set of locations>;
  @*/
T m(T1 p1, ..., Tn pn)
```

- keyword **assignable** defines a list of heap locations (static/instance fields, array elements) the method is allowed to change (locations not listed must be unchanged after termination of `m` or must not have existed in the pre-state).

  Special values:
  - **`assignable`** `\everything`: method may cause arbitrary side effects
  - **`assignable`** `\nothing`: method must not change any location existing in the pre-state (may create new objects)
  - **`assignable`** `\strictly_nothing`: method must neither change any location nor create new objects

Recapture of Previous Lecture

Focus: Basic JML expressions and normal behavior specification of methods

```
/*@ <visibility modifier> normal_behavior
  @ requires P;
  @ ensures Q;
  @ assignable <set of locations>;
  @*/
T m(T1 p1, ..., Tn pn)
```

- for a method to adhere to the above normal behavior specification, it must guarantee that if called in a state (the pre-state) which satisfies the precondition P that it will
  - terminate normally,
  - terminate in a state (the post-state) satisfying Q and
  - in the post-state only the locations allowed by the assignable clause might have a different value than in the pre-state

JML Expressions (so far)

TECHNISCHE UNIVERSITÄT DARMSTADT

Definition (JML Expressions)

- Each side-effect free JAVA expression is a JML expression
- If *E* is a side-effect free JAVA expression, then `\old(`*E*`)` is a JML expression
- If `a` and `b` are **`boolean`** JML expressions, `x` is a variable of type `t`:
  - `!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` is true")
  - `(\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` is true")
  - `(\exists t x; a; b)` ("there exists an `x` of type `t` fulfilling `a`, such that `b` is true")

  are also **`boolean`** JML expressions.

Range Predicates

TECHNISCHE UNIVERSITÄT DARMSTADT

Definition (Range predicate)

In the JML expressions `(\forall t x; a; b)` and `(\exists t x; a; b)` the **`boolean`** **`a`** is called range predicate.

Range predicates are syntactic sugar for standard FOL quantifiers:

`(\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

TECHNISCHE UNIVERSITÄT DARMSTADT

Range predicates used to restrict range of `x` further than to its type `t`

Example

“Array `a` is sorted between indices 0 and 9”:

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

Using Quantified JML Expressions

TECHNISCHE UNIVERSITÄT DARMSTADT

- An array `int a` contains only non-negative elements

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

- The variable `m` holds a maximal element of array `a`

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

Is this sufficient? Need in addition:

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

HealthTracker: StepCounter

TECHNISCHE
UNIVERSITÄT
DARMSTADT

How to specify the behavior of methods in an interface? Use pure methods.

```
/
 * A step counter tracks the number of steps until a
 * {@link reset()} occurs. After resetting, counting restarts
 * with 0 steps. Call {@link incSteps(int)} to update counter.
 * Walked distance is computed using the total number of steps
 * and the step size. */
public interface StepCounter {
  public int getStepsTotal();
  public void reset();
  public void incSteps(int p_inc);

  public int getStepSize();
  public int getDistance();
  public void setStepSize(int size);
}
```

Pure Methods: Using Queries in Specifications

Method `get<XXX>()` pure methods in interface (others have side-effects), e.g.,

```
public /*@ pure @*/ int getStepsTotal();
```

Can be used to express the behavior of other methods, e.g.,

```
/*@ public normal_behavior
  @ requires true; // <- can be ommitted
  @ ensures getStepsTotal() == 0;  @*/
public void reset();

/*@ public normal_behavior
  @ requires p_inc >= 0;
  @ ensures getStepsTotal()==\old(getStepsTotal()) + p_inc; @*/
public void incSteps(int p_inc);
```

All methods can be completely specified using `getStepsTotal()`. `getStepSize()`

Pure Methods: Using Queries in Specifications

TECHNISCHE UNIVERSITÄT DARMSTADT

Usage of pure methods enables us to

- specify the behavior of methods declared in interfaces
- reduce the need for **spec_public** fields in specifications (**spec_public** violates hiding in specifications)
- provide variability/flexibility for the actual implementation

What about **assignable**?

- missing assignable clause means **assignable \everything;** (method may have arbitrary side-effects)

Limit of this approach. For a solution, see a later lecture.

(Exercise: Use queries to simplify the specification of classes `HealthTracker` and `Category` of previous lecture. Reduce/Eliminate usage of **spec_public** and usage of fields in method specifications.)

Capturing Object Inherent Design Properties

```
public interface StepCounter {
  public /*@ pure @*/ int getStepsTotal();
  public void reset();
  public void incSteps(int p_inc);

  public /*@ pure @*/ int getStepSize();
  public /*@ pure @*/ int getDistance();
  public void setStepSize(int size);
}
```

Which other properties would we expect?

For instance,

- step count non-negative
- step size positive (strictly greater than zero)

Capturing Object Inherent Design Properties

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface StepCounter {
  public /*@ pure @*/ int getStepsTotal();
  public void reset();
  public void incSteps(int p_inc);

  public /*@ pure @*/ int getStepSize();
  public /*@ pure @*/ int getDistance();
  public void setStepSize(int size);
}
```

Which other properties would we expect?

For instance,

- step count non-negative
- step size positive (strictly greater than zero)

These properties should hold at any time. Clients should be able to rely on them.

Capturing Object Inherent Design Properties

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface StepCounter {
  public /*@ pure @*/ int getStepsTotal();
  public void reset();
  public void incSteps(int p_inc);

  public /*@ pure @*/ int getStepSize();
  public /*@ pure @*/ int getDistance();
  public void setStepSize(int size);
}
```

Which other properties would we expect?

For instance,

- step count non-negative
- step size positive (strictly greater than zero)

One possibility: Add them as pre- and postcondition to each method.

Object State Properties as Pre-/Post Pairs

TECHNISCHE UNIVERSITÄT DARMSTADT

Adding object properties as pre-/post pairs:

```
/*@ public normal_behavior
  @ requires getStepsTotal() >= 0 && getStepSize() >= 0;
  @ ensures getStepsTotal() == 0 &&
  @   getStepsTotal() >= 0 && getStepSize() >= 0;
  @*/
public void reset();

/*@ public normal_behavior
  @ requires p_inc >= 0 &&
  @   getStepsTotal() >= 0 && getStepSize() >= 0;
  @ ensures getStepsTotal() == \old(getStepsTotal()) + p_inc &&
  @   getStepsTotal() >= 0 && getStepSize() >= 0
  @*/
public void incSteps(int p_inc);
```

Object State Properties as Pre-/Post Pairs

TECHNISCHE UNIVERSITÄT DARMSTADT

Adding object properties as pre-/post pairs:

```
/*@ public normal_behavior
  @ requires getStepsTotal() >= 0 && getStepSize() >= 0;
  @ ensures getStepsTotal() == 0 &&
  @                                                       0;
  @*/
publi

/*@ pu
  @ re
  @                                                       ;
  @ en                                           ()) + p_inc &&
  @
  @*/
public void incSteps(int p_inc);
```

Problems:

- clutters specification
- must be added to all additional methods of implementing classes

Better solution: Class Level Specifications

Class Level Specifications

How to specify constraints on state of a class?

Class level specifications place restrictions on the object state

Kinds of Class Level Specifications in JML

- class **invariants** (or synonym: object invariants)
- (**initially** clauses)
- (history **constraints**)

We focus on class invariants.

From where do class invariants come?

- Modeled reality (e.g., there is no such thing as negative steps)
- Consistency of redundant data representations (e.g., caching)
- Restrictions for efficiency (e.g., maintaining sortedness)

Class Invariants: Step Counter Cnt'd

```
public interface StepCounter {
  //@ public instance invariant getStepsTotal() >= 0;
  //@ public instance invariant getStepSize() >= 0;
  //@ public instance invariant
  //@       getStepSize() * getStepsTotal() == getDistance();

  public /*@ pure @*/ int getStepsTotal();
  ...
  public /*@ pure @*/ int getStepSize();
  public /*@ pure @*/ int getDistance();
  public void setStepSize(int size);
}
```

Specifying State Constraints

So far: JML used to specify (local) method behavior

How to specify constraints on state of a class?

- Consistency of redundant data representations (e.g., caching)
- Restrictions for efficiency (e.g., maintaining sortedness)

Constraints on state are global: all methods must preserve them

Static vs. instance invariants

Instance invariants, on the other hand, can access the **this**-reference. For instance,

```
public class SimpleStepCounter implements StepCounter{
  private int stepSize;
  //@ private instance invariant this.stepSize >=0;
  ...
}
```

Static invariants do not have access to the **this**-reference.

```
public class SimpleStepCounter implements StepCounter{
  private static int MAX_STEP_SIZE;
  //@ private static invariant MAX_STEP_SIZE >=0;
  ...
}
```

Static vs. instance invariants: Defaults

TECHNISCHE UNIVERSITÄT DARMSTADT

If no modifier **instance** or **static** is given, invariants

- in classes are by default instance invariants
- in interfaces are by default static invariants

System Correctness

When do we call a software system correct?

In other words, when does

- an implementation adhere to a specification, e.g., where and when does it need to
  - establish,
  - preserve and
  - assume invariants (and which of them)
- what are the global guarantees of a *correct* implementation, e.g.,
  - does it impose a closed-world or open-world assumption (modularity)

Non-trivial research question.

Class Invariants: Intuition, Notions & Scope

Basic Intention/Intuition: Class invariants must be

- established by
  - the constructor (instance invariants) and
  - static initialisation (static invariants)
- preserved by all non-helper methods
  - assumed in prestate (i.e., invariants are implicit preconditions)
  - ensured in poststate (i.e., invariants are implicit postconditions)
  - they can be violated during method execution

When must an implementation ensure that an invariant holds?

- Method invocation
- Method termination (normal or abrupt termination)

Scope of invariants

Invariants are written local, but are actually system wide properties.

Consequently: Invariants must not be violated by any object of any class.

The JML modifier: `helper`

JML helper methods

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

neither assume nor ensure any invariant.

Pragmatics & Usage examples of helper methods

Helper methods are almost always **private**.

Usage examples comprise:

- inside constructors, where invariants have not yet been established
- structural changes of a linked data structure might intermediately violate invariants (e.g., rotation methods in Red-Black trees)

Visible State Semantics for Class Invariants

TECHNISCHE UNIVERSITÄT DARMSTADT

Visible State Semantics (JML standard semantics)

- Instance invariants for an object *o* must be satisfied in all states visible for *o*
- Static invariants for a type *T* must be satisfied in all states visible for *T*

Definition (Visible State for an Object *o* (JML Ref. Manual, Sec. 8.2))

A visible state for an object *o* is a state that occurs at one of these moments:

- at the end of a non-helper constructor invocation (invoc.) that is initializing *o*,
- at the beginning or end of a non-helper
  - instance method invoc. with *o* as the receiver,
  - static method invoc. for a method in *o*'s class or some of its superclasses, or
- when no constructor/instance method invoc. with *o* as receiver, or static method invoc. for a method in *o*'s or some of its superclass is in progress.

Visible State Semantics for Class Invariants

TECHNISCHE UNIVERSITÄT DARMSTADT

Definition (Visible State for a Type T (JML Ref. Manual, Sec. 8.2))

A visible state for a type T is a state that

- occurs after static initialization for T is complete and
- it is a visible state for some object that has type T.

Note: Objects of subtypes of type T also have type T.

Visible State Semantics for Class Invariants: Example

TECHNISCHE UNIVERSITÄT DARMSTADT

Assume two different objects *o*, *u* of class `SimpleStepCounter`

Visible state for object *o*, *u*?

```
m() { // neither a method of o or u
  o.mergeCounters(u);
  ...
}
                         ○                      o,u
void mergeCounters(StepCounter p_other) {
                         ○                      u
  int otherSteps = p_other.getStepsTotal();
                         ○                      u
                         ○                      o, u
  int ownSteps = this.getStepsTotal();
                         ○                      o,u
  ...;
}
                         ○                      o,u
```

Visible State Semantics for Class Invariants: Example

Assume two different objects *o*, *u* of class `SimpleStepCounter`

```
m() { // neither a method of o or u
  o.mergeCounters(u);
  ...
}
```

Visible state for object *o*, *u*?

∘ o,u

```
void mergeCounters(StepCounter p_other) {
```

∘ u

```
  int otherSteps = p_other.getStepsTotal();
```

```
  int ownSte
```

For all other objects (except the one executing `m`) all states of the considered states are visible states

```
  ...;
}
```

∘ o,u

Visible State Semantics for Class Invariants

TECHNISCHE UNIVERSITÄT DARMSTADT

JML ensures invariants by implicitly requiring from each non-helper method (or constructor) m

1. to assume the
   - instance invariants of all objects o, for which m's pre-state is a visible state for o
   - static invariants of all types T, for which m's pre-state is a visible state for T
2. to establish the
   - instance invariants of all objects o, for which m's post-state is a visible state for o
   - static invariants of all types T, for which m's post-state is a visible state for T

Visible State Semantics: Problems

The visible state semantics has some severe short-comings:

- non-modular: changing any invariant requires to reverify (almost) all methods
- too strong restrictions on methods/types and their clients

```
public class Pair {
  private Object first;
  private Object second;

  public Pair(Object p_first, Object p_second) { ...  }
  public /*@ pure @*/ Object getFirst() { return first; }
  public /*@ pure @*/ Object getSecond() { return second; }
}
```

Using visible state semantics, `getFirst`, `getSecond`— assume and ensure (among others) invariants of contained elements. Which is unnecessary as behavior of `Pair` does not rely on any properties of the contained objects.

Semantics of Class Invariants: JML*

TECHNISCHE UNIVERSITÄT DARMSTADT

We use JML*: A JML variant with (among others) a different invariants semantics

(The basic idea is taken from Spec# and called the Boogie-Methodology; the realisation differs)

Idea: Give responsibility where invariants are assume or ensured back to specifier

JML* Keyword: `\invariant_for(o)`

`\invariant_for(o)` is a Boolean JML expression which evaluates to true iff. all accessible instance invariants of *o* are satisfied.

Example

```
/*@ public normal_behavior
  @ requires \invariant_for(this) && \invariant_for(key);
  @ ensures \invariant_for(this) && \invariant_for(key); @*/
public void put(Object key, Object value) { ... }
```

specifies that `put` assumes and ensures the invariants of `this` and `key`

Semantics of Class Invariants: JML*

TECHNISCHE UNIVERSITÄT DARMSTADT

We use JML*: A JML variant with (among others) a different invariants semantics

(The basic idea is taken from Spec# and called the Boogie-Methodology; the realisation differs)

Idea: Give responsibility where invariants are assume or ensured back to specifier

JM

\in ... es to true iff. all

acc

For non-helper methods \invariant_for(this) implicitly added to pre- and postconditions!

For static invariants: \static_invariant_for(TypeRef)

Ex

```
/*@ public normal_behavior
  @ requires \invariant_for(this) && \invariant_for(key);
  @ ensures \invariant_for(this) && \invariant_for(key); @*/
public void put(Object key, Object value) { ... }
```

specifies that put assumes and ensures the invariants of this and key

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` (this is the default)

non_null: Examples

TECHNISCHE UNIVERSITÄT DARMSTADT

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

Implicit invariant `public invariant username != null;` added to class for fields of reference type

```
public void addCategory(/*@ non_null @*/ Category p_category)
```

Implicit precondition `requires p_category != null;`
added to each specification case of `addCategory`

```
public /*@ non_null @*/ Category findCategoryById(int)()
```

Implicit postcondition `ensures \result != null;`
added to each specification case of `findCategoryById()`

`non_null` is default in JML:
all of the above **`non_null`**'s are redundant

`nullable`: Examples

TECHNISCHE UNIVERSITÄT DARMSTADT

Prevent `non_null` pre/post-conditions, invariants: `nullable`

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

No implicit invariant added, `username` might have value `null`

- Some of our earlier examples need `nullable` to work properly, e.g.:

```
private /*@ nullable @*/ Category findCategoryById(int p_id);
```

LinkedList: non_null or nullable?

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

Consequence of default non_null in JML

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

Repair so that the list can be finite:

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

Final Remarks on `non_null` and `nullable`

TECHNISCHE UNIVERSITÄT DARMSTADT

`non_null` as default in JML only since a few years

Older JML tutorials/articles might use `nullable`-by-default semantics

Pitfall!

```
/*@ non_null @*/ Category[] category;
```

is not the same as:

```
//@ invariant category != null;
/*@ nullable @*/ Category[] category;
```

The first adds implicitly:

```
(\forall int i; i>=0 && i<category.length; category[i] != null)
```

I.e., requires `non_null` of all array elements!