Formal Specification and Verification of Object-Oriented Programs

JML: Invariants, Behavioral Subtyping & Exceptional Behavior

TECHNISCHE UNIVERSITÄT DARMSTADT

Recapture of Previous Lecture

How to specify constraints on state of an object?

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)

Recapture of Previous Lecture: Semantics of Class Invariants

TECHNISCHE UNIVERSITÄT DARMSTADT

Discussion about JML's standard visible state semantics and its severe drawbacks

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

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`

Recapture of Previous Lecture: Semantics of Class Invariants

Discussion about JML's standard visible state semantics and its severe drawbacks

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

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

JML* Ke

\invaria ue iff. all

accessible

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

For static invariants: \static_invariant_for(TypeRef)

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`

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

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`

`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:

```
//@ private invariant category != null;
private /*@ 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!

Supertype Abstraction: Motivation


```
n(T e) { ...  res =  e.m()  ... }
```

What does the developer expect?

`e.m()` behaves in non-surprising ways when overwritten.

More precise: Contract of n() should hold independent of dynamic type of e

How to ensure that `n()` 's contract holds in presence of dynamic dispatch?

Two possibilities:

1. check for all implementations of `m()` or
2. assume only contract of static type of `T`

Supertype Abstraction: Motivation

TECHNISCHE UNIVERSITÄT DARMSTADT

How to ensure that `n()` 's contract holds in presence of dynamic dispatch?

Two possibilities:

1. check for all implementations of `m()` or
2. assume only contract of static type of `T`

Checking for all implementations is not modular need to reverify/-check all calling sites of `m()` as soon as

- one of its implementation changes or
- new subtype is added which overwrites `m()`

Supertype Abstraction: Motivation

TECHNISCHE UNIVERSITÄT DARMSTADT

How to ensure that `n()` 's contract holds in presence of dynamic dispatch?

Two possibilities:

1. check for all implementations of `m()` or
2. assume only contract of static type of `T`

Assuming static contract of `T` is called supertype abstraction

- modular, but
- only sound (correct), in presence of behavioral subtyping

Behavioral Subtyping

Several definition of behavioral subtypes have been stated

Most famous one:

Liskov's Substitution Principle (Barbara Liskov)

Liskov's Substitution Principle


```
class T {
      V m() { ... }
}
```

```
class S extends T {
      V m() { ... }
}
```

```
class U {   n(T e) { ...  res =  e.m()  ... } }
```

Liskov's Substitution Principle (LSP)

- object invariants: $inv_S \Rightarrow inv_T$
  ("invariants of subtype imply the invariants of the supertype")

Liskov's Substitution Principle

```
class T {                          class S extends T {
      V m() { ... }                      V m() { ... }
}                                  }

class U {   n(T e) { ...  res =  e.m()  ... } }
```

Liskov's Substitution Principle (LSP)

- method specifications. Let (pre_m^T, $post_m^T$) be the psec. of m in supertype and (pre_m^S, $post_m^S$) the specification of the overwriting method in subtype S
  - $pre_m^T \Rightarrow pre_m^S$
    (ensures that every valid prestate for m as defined in T (write $m() : T$) is also a valid prestate for the overwriting method)
  - $post_m^S \Rightarrow post_m^T$
    (ensures that every property which holds in a poststate of $m() : T$ (under assumption m is called in a valid prestate) holds also in poststate of $m() : S$)

Applicability of LSP in Practice

TECHNISCHE UNIVERSITÄT DARMSTADT

Provable that LSP is sufficient for correctness of supertype abstraction.

But:

```
class Account {
 int balance;
 /*@ normal_behavior
   @ requires amount >= 0;
   @ ensures
   @  balance>=\old(balance);
   @*/
 void update(int amount)
}
```

```
class DebAccount extends Account {
 /*@ normal_behavior
   @ requires true;
   @ ensures
   @  amount>=0 ==>
   @       balance>=\old(balance);
   @ ensures
   @  amount<0 ==>
   @       balance<\old(balance);
   @*/
  void update(int amount)
}
```

Applicability of LSP in Practice

TECHNISCHE UNIVERSITÄT DARMSTADT

Provable that LSP is sufficient for correctness of supertype abstraction.

But:

```
class DebAccount extends Account {
```

Observation:

LSP is violated by many programs in practice.

Good news:

LSP is unnecessarily strong

```
  @ requires amount >= 0;
  @ ensures
  @  balance>=\old(balance);
  @*/
 void update(int amount)
}
```

```
  @       balance>=\old(balance);
  @ ensures
  @  amount<0 ==>
  @       balance<\old(balance);
  @*/
 void update(int amount)
}
```

Applicability of LSP in Practice

TECHNISCHE UNIVERSITÄT DARMSTADT

Provable that LSP is sufficient for correctness of supertype abstraction.

But:

For correctness of supertype abstraction weaker (more flexible) definitions of behavioral subtyping are sufficient.

```
class Account {
 int balance;
 /*@ normal_beh
   @ requires amount >= 0;
   @ ensures
   @  balance>=\old(balance);
   @*/
 void update(int amount)
}
```

```
ends Account {

 @  amount>=0 ==>
 @       balance>=\old(balance);
 @ ensures
 @  amount<0 ==>
 @       balance<\old(balance);
 @*/
void update(int amount)
}
```

A Weaker Definition of Behavioral Subtyping

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class T {
      V m() { ... }
}
```

```
class S extends T {
      V m() { ... }
}
```

```
class U {   n(T e) { ...  res =  e.m()  ... } }
```

Improved (and weaker than LSP) definition of behavioral subtyping:

If the following holds

$$\texttt{\textbackslash old}(pre_m^T) \,\&\&\, post_m^S \Rightarrow post_m^T$$

then

supertype abstraction is sound

Ensuring Behavioral Subtyping by Inheritance

TECHNISCHE UNIVERSITÄT DARMSTADT

All JML contracts, i.e.

- specification cases
- class invariants

are inherited from superclasses to subclasses

A class must fulfill all contracts of all its superclasses

Subclasses may add specification cases to those of superclasses:

```
/*@ also
  @
  @ <specification-case-specific-to-subclass>
  @*/
  public void method () { ... }
```

Initially Clauses

TECHNISCHE UNIVERSITÄT DARMSTADT

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

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

Initially clauses are

- additional postconditions to constructors.
- inherited by subclasses (in contrast to constructor specification cases)

History Constraints

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface StepCounter {
  //@ public invariant getStepsTotal() >= 0;
  //@ public invariant getStepSize() >= 0;
  //@ public constraint \old(getStepsTotal()) <= getStepsTotal();

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

History constraints

- relate two successive states of an object (implicit postcondition for all methods)
- inherited by subclasses
- tricky to use (almost no tool support): as defined relation
  - must ensure reflexivity (otherwise spec. would forbid pure methods)
  - should usually ensure transitivity

Methods throwing Exceptions

TECHNISCHE UNIVERSITÄT DARMSTADT

Previous lecture: all specification cases were about **normal_behavior**

```
/*@
 <spec-case1: Max Reached>
 also
 <spec-case2: Category of same id present>
 also
 <spec-case3: Add category>
 @*/
public boolean addCategory (Category p_category) { ... }
```

We want now to specify that the method should throw an `IllegalArgumentException`, if **null** is passed as argument.

Specifying Exceptional Behavior of Methods

`normal_behavior` specification case

Assume precondition (**`requires`** clause) *P* fulfilled

- Forbids method to throw exception when pre-state satisfies *P*

`exceptional_behavior` specification case

Assume precondition (**`requires`** clause) *P* fulfilled

- Requires method to throw exception when pre-state satisfies *P*
- Keyword **`signals`** specifies *post-state*,
  depending on type of thrown exception
- Keyword **`signals_only`** specifies type of thrown exception

JML specifications must separate normal/exceptional specification cases by suitable preconditions

Specifying Exceptional Behavior of `addCategory(Category)`

```
/*@ <spec-case1> also <spec-case2> also <spec-case3> also
  @ public exceptional_behavior
  @ requires p_category==null;
  @ signals_only IllegalArgumentException;
  @ signals (IllegalArgumentException e)
  @           e.getMessage().equals("Null not allowed."); @*/
public boolean addCategory (/*@ nullable @*/ Category p_category)
```

Meaning (ommitting invariants, see later)

When `p_category==null` holds in pre-state ...

- An exception must be thrown (`exceptional_behavior`)
- This can only be an `IllegalArgumentException` (`signals_only`)
- In its final state the method must ensure
  `e.getMessage().equals("Null not allowed.")` (`signals`)

`signals_only` Clause: General Case

An exceptional specification case can have at most one clause of the form

$$\texttt{signals\_only}\ \mathtt{E}_1,\ldots,\mathtt{E}_n;$$

where $\mathtt{E}_1,\ldots,\mathtt{E}_n$ are exception types

The thrown exception must have type $\mathtt{E}_1$ or $\cdots$ or $\mathtt{E}_n$

`signals_only` Clause: General Case

By default (i.e., if not explicitly stated) `signals_only` contains all exceptions of a method's throws clause as well as `RuntimeException` and `Error`.

```
/*@ public exceptional_behavior
  @ requires P;
  @*/
void parse(InputStream is) throws RecognitionException,SemanticException
```

is equivalent to

```
/*@ public exceptional_behavior
  @ requires P;
  @ signals_only RecognitionException, SemanticException,
  @     RuntimeException, Error;
  @*/
void parse(InputStream is) throws RecognitionException,SemanticException
```

`signals` Clause: General Case

TECHNISCHE UNIVERSITÄT DARMSTADT

An exceptional specification case can have several clauses of the form

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

where `E` is an exception type, `b` is a boolean JML expression

If an exception of type `E` is thrown, then `b` holds in the post-state.

In the post-state of `non_helper` methods, `\invariant_for(this)` must hold as well.

Non-Termination

TECHNISCHE UNIVERSITÄT DARMSTADT

By default, both:

- `normal_behavior`
- `exceptional_behavior`

specification cases enforce termination

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

```
diverges true;
```

If the precondition of the specification case holds in the pre-state, then the method may or may not terminate

General Behaviour Specification Case

TECHNISCHE UNIVERSITÄT DARMSTADT

Complete Behavior Specification Case

```
behavior
  forall T1 x1; ...  forall Tn xn;
  old U1 y1 = F1; ...  old Uk yk = Fk;
  requires P;
  measured_by Mbe if Mbp;
  diverges D;
  when W;
  accessible R;
  assignable A;
  callable p1(...), ..., pl(...);
  captures Z;
  ensures Q;
  signals_only E1, ..., Eo;
  signals (E e) S;
  working_space Wse if Wsp;
  duration De if Dp;
```

gray	not in this course
green	in one way or the other already seen
red	future topic of this course

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

TECHNISCHE UNIVERSITÄT DARMSTADT

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

General Behaviour Specification Case

TECHNISCHE UNIVERSITÄT DARMSTADT

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) post-state: All of the following items must hold
  - only heap locations ...
  - if *m* terminated normally then in its post-state, property *Q* holds (default: Q=**true**)
  - if *m* terminated abruptly then with
    - one of the exception listed in **signals_only** (default: all exceptions of *m*'s throws declaration + `RuntimeException` and `Error`) and
    - for matching **signals** clauses, the exceptional postcondition *S* holds (default: no clause)

General Behaviour Specification Case

TECHNISCHE UNIVERSITÄT DARMSTADT

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) post-state: All of the following items must hold
  - ...
  - `\invariant_for(this)` must hold (no matter whether normal or abrupt termination) for `non_helper` methods

Desugaring:
Normal Behavior and Exceptional Behavior

TECHNISCHE UNIVERSITÄT DARMSTADT

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

Normal Behavior Case

- defaults for signals to **signals** `(Throwable e)`**false**; and
- forbids overwriting of **signals** and **signals_only**

Exceptional Behavior Case

- defaults for **ensures** to false and
- forbids overwriting of **ensures**

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