Formale Grundlagen der Informatik 3

Java Modeling Language, Part II

Reiner Hähnle

14 January 2014

Recapitulation

Java Modeling Language (JML)

- A formal specification language tailored to JAVA
- Specifications appear as structured comments in `.java` files
- JML expressions contain pure JAVA expressions and quantified formulas (with optional range predicates)
- Granularity of specifications are OO units: classes and methods
- Specification methodology follows the contract paradigm
  - Callee ensures outcome if caller fulfills requirements
- Structure of JML method contracts
  - Visibility, type of behavior (normal, exceptional)
  - One or more specification cases
  - Each case has precondition, postcondition, assignable locations
- Use **`\old()`** to access values in pre-state

Example: Specifying `LimitedIntegerSet`

```
public class LimitedIntegerSet {
  public final int limit;
  private int a[];      // holds the set elements
  private int size = 0; // current cardinality

  public LimitedIntegerSet(int limit) {
    this.limit = limit;
    this.a = new int[limit];
  }

  public boolean add(int elem) { ... }

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

  public boolean contains(int elem) { ... }
}
```

Prerequisites: Adding Specification Modifiers

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

  public LimitedIntegerSet(int limit) {
    this.limit = limit;
    this.a = new int[limit];
  }

  public boolean add(int elem) { ... }

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

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

Result Values

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

- Method has no effect on the state, in particular, no exceptions
- We need to specify the result value

In postconditions `\result` refers to the return value of a method

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

Specifying add() (spec-case1) new element is added

```
/*@ public normal_behavior
  @ requires size < limit && !contains(elem);
  @ ensures \result;
  @ ensures contains(elem); // call of pure method
  @ 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;
  @ ensures (\forall int e;
  @                  contains(e) <==> \old(contains(e)));
  @ ensures size == \old(size);
  @*/
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;
  @ assignable \nothing;
  @ // Does this solution make any difference?
  @
  @*/
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;
  @ assignable \nothing;
  @ // Does this solution make any difference?
  @ // 1st solution: ok to reorder a, change other fields
  @*/
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) { ... }
```

- Can you explain in words what the different ensures clauses mean?

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

Consider LimitedSortedIntegerSet

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

  public LimitedSortedIntegerSet(int limit) {
    this.limit = limit;
    this.a = new int[limit];
  }

  public boolean add(int elem) { ... }

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

  public boolean contains(int elem) { ... }
}
```

Consequence of Sortedness for Implementation

Method `contains()`

- Assume sortedness in pre-state
- Implementation can employ binary search (logarithmic complexity)

Method `add()`

- Assume sortedness in pre-state
- Binary search for first index with bigger element, insert just before it
- Must maintain sortedness in post-state

Method `remove()`

(accordingly)

Specifying Sortedness with JML

Express sortedness over the fields of the class

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

Sortedness as JML expression

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

(what's the value of this when `size < 2`?)

Where in the specification does the red expression go?

Specifying Sorted `contains()`

Assume sortedness of pre-state

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

- `contains()` is pure ⇒ sortedness of post-state trivially ensured

Specifying Sorted `remove()`

Assume sortedness of pre-state — Ensure sortedness of post-state

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

Factoring out Sortedness

Need to do the same for both specification cases of `add()` ...
Need to do the same for both specification cases of `remove()` ...
Need to add sortedness as postcondition of constructor ...
⇒ Adding sortedness clutters method contracts

JML Class Invariant: specify global state constraints

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;

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

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

  // constructor and methods,
  // without sortedness in pre/post-conditions
}
```

JML Class Invariant Cont'd

- JML **`invariant`** declaration may appear anywhere in class (contrast: method contract must be in front of its method)
- Convention: place class invariant in front of fields it talks about

Instance vs. Static Invariants

Instance invariants

Can refer to instance fields of `this` object

- unqualified, e.g., `size`, or qualified with **`this`**, e.g., **`this`**`.size`)

JML syntax: **`instance invariant`**

Static invariants

Cannot refer to instance fields of `this` object

JML syntax: **`static invariant`**

Instance and static invariants

Can refer to

- static fields
- instance fields via quantifying over explicit reference, e.g., `o.size`

In classes: **instance is default** (static in interfaces)

When **`instance`** or **`static`** is omitted ⇒ default is used!

Semantics of JML/KeY Class Invariants

Intuition

For each method `m()` in a class `C`:

- For each class invariant I of `C` (including super):
  - Add I for caller object to pre-/postcondition of `m()`'s contract
  - If `m()` is a constructor, it must establish I for the new object

Invariants need not hold during execution of a method

Static JML Invariant Example

Recall the banking card example from the previous lecture:

```
public class BankCard {

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

  private /*@ spec_public @*/ int cardNumber;

  // rest of class
}
```

Recall Specification of `enterPIN()`

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

/*@
 <spec-case1: PIN correct>
 also
 <spec-case2: PIN incorrect, wrong PIN counter below 2>
 also
 <spec-case3: PIN incorrect, wrong PIN, card confiscated>
 @*/
public void enterPIN (int pin) { ... }
```

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

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 `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) { ... }
```

Meaning

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

- An exception must be thrown (`exceptional_behavior`)
- This can only be an `ATMException` (`signals_only`)
- In its final state the method must ensure `!customerAuthenticated` (`signals`)

`signals_only` Clause: General Case

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

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

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` Clause: General Case

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

Non-Termination

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

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

```
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` 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 name;
```

No implicit invariant added, `name` might have value **`null`**

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

```
private /*@ spec_public nullable @*/
                        BankCard insertedCard = null;
```

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 @*/ Object[] a;
```

is not the same as:

```
/*@ nullable @*/ Object[] a; //@ invariant a != null;
```

The first also implicitly adds:

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

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

JML and Inheritance

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 () { ... }
```

JML Tools

- Many tools support JML (http://www.eecs.ucf.edu/~leavens/JML/download.shtml)
  - Most support only a fragment of JML
- The KeY system contains a JML parser for the fragment it supports

Literature for this Lecture

Essential Reading

KeY Book Andreas Roth & Peter H. Schmitt: Formal Specification. Chapter 5, Sections 5.1, 5.3, In: B. Beckert, R. Hähnle, and P. Schmitt, eds. *Verification of Object-Oriented Software: The KeY Approach*, vol 4334 of *LNCS*. Springer, 2006.

http://link.springer.com/book/10.1007/978-3-540-69061-0/

Further Reading

http://www.eecs.ucf.edu/~leavens/JML/documentation.shtml

JML Reference Manual G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, and J. Kiniry. *JML Reference Manual*, July 2011

JML Tutorial G. T. Leavens, Y. Cheon. *Design by Contract with JML*

JML Overview G. T. Leavens, A. L. Baker, and C. Ruby. *JML: A Notation for Detailed Design*