JML and BCSL

Ando Saabas

IoC theory seminar
23.03.06

Outline

- Design by contract
- JML
- JML tools
- BCSL

Design by contract

- A program (a class) and its clients should have a “contract" with each other
  - The client must guarantee certain conditions before calling a method defined by the class
  - In return the class guarantees certain properties that will hold after the call
- One can avoid constantly checking arguments
- Makes it easier to assign blame

JML

- JML is a formal behavioral interface specification language for Java.
- Started at Iowa State University by the group of Gary Leavens
- It allows one to specify both the syntactic interface of Java code and its behavior.
- JML uses Java's expression syntax to write the predicates used in assertions
  - Makes it easer easier for programmers to learn JML
- Java's expressions are extended with various specification constructs, such as quantifiers

Annotations

- JML specifications can be
  - written in separate files
  - contained in annotations, which are comments like:

```
//@ …
```

or

```
/*@ …
  @ …
  @*/
```

A contract

```
public class IntMathOps {
  /*@ public normal_behavior
    @   requires y >= 0;
    @   assignable \nothing;
    @   ensures 0 <= \result
    @   && \result * \result <= y
    @   && (y < (\result + 1) * (\result + 1));
    @*/
  public static int isqrt(int y) {
       return (int) Math.sqrt(y);
  }
}
```

	Obligations	Rights
User	Passes non-negative number	Gets square root approximation
Class	Computes and returns square root	Assumes argument is non-negative

Class and Interface Specifications

- *Invariants.* A property that should always be true of an object's state (when control is not inside the object's methods).
- Invariants allow you to define:
  - Acceptable states of an object, and
  - Consistency of an object's state.

```
//@ public invariant !name.equals(“”) &&
  weight >= 0;
```

Class and Interface Specifications

- *Model Fields*
  - Do not have to have an implementation.
  - For purposes of the specification, it is treated like any other Java field.
  - **`represents`** clause can be used to say how a model field is related to an actual field
- *History constraints* – states how values can change between earlier and later publicly-visible states
  - **`public instance constraint MAX_SIZE == \old(MAX_SIZE);`**

Method specifications

- Pre- and postconditions
  - **`requires`** and **`ensures`**

    ```
    requires !stack.isempty();
    ensures \result == stack.first()
    ```
- Assignable clause
  - Gives frame conditions: allows to assign only to locations given in the assignable clause.
  - **`assignable stack;`**

Method specifications

- **`normal_behavior`** keyword notes that method should finish normally
- **`behavior`** keyword notes that there can be an exception thrown
- **`exceptional_behavior`** states that the method must always terminate with an exception
- **`signals`** clause can be used to describe under what condition an exception can be thrown

- **`normal_behaviour == signals (java.lang.Exception) false`**
- **`exceptional_behaviour == ensures false`**

Semantics of method specifications

- A method must be called in a state (prestate) where the method's precondition is satisfied
- If a method is called in a proper pre-state, then
  - if the method terminates normally (without throwing an exception), then in the termination state (normal poststate), its normal postcondition must be satisfied.
  - If the method terminates by throwing an exception, then in the termination state (exceptional post-state), then the exceptional post-state must satisfy the corresponding exceptional postconditions

Example

```
public interface BoundedThing {
  //@ public model instance int MAX_SIZE;
  //@ public model instance int size;

  /*@
   public instance invariant MAX_SIZE > 0;
   public instance invariant  0 <= size && size <= MAX_SIZE;
   public instance constraint MAX_SIZE == \old(MAX_SIZE);@*/

  /*@
   public normal_behavior
   ensures \result == MAX_SIZE;    @*/
   public /*@ pure @*/ int getSizeLimit();
  /*@
   public normal_behavior
   ensures \result <==> size == 0;      @*/
   public /*@ pure @*/ boolean isEmpty();
```

```
/*@  public normal_behavior
     ensures \result <==> size == MAX_SIZE;
  @*/
    public /*@ pure @*/ boolean isFull();

    /*@ also
         public behavior
           assignable \nothing;
           ensures \result instanceof BoundedThing
               && size == ((BoundedThing)\result).size;
           signals_only CloneNotSupportedException;
      @*/
    public Object clone ()
       throws CloneNotSupportedException;
}
```

Method specifications

- Purity of methods
  - Specified with the modifier **`pure`**
  - Refines the following:

```
behavior
  assignable \nothing
```

  - It must also be provably terminating
- Loop variant and invariant:
  - **`maintaining`** *predicate*
  - **`decreasing`** *expression*

```
public abstract class SumArrayLoop {
 //@ requires a != null;
 //@ requires (\sum int j; 0 <= j && j < a.length; a[j]) <=
   Long.MAX_VALUE;
 //@ requires (\sum int j; 0 <= j && j < a.length; a[j]) >=
   Long.MIN_VALUE;
 //@ assignable \nothing;
 //@ ensures \result == (\sum int j; 0 <= j && j < a.length; a[j]);

  public static long sumArray(int [] a) {
    long sum = 0;
    int i = a.length;
    /*@ maintaining -1 <= i && i <= a.length;
      @ maintaining sum
      @    == (\sum int j; i <= j && 0 <= j && j < a.length; a[j]);
      @ decreasing i; @*/
    while (--i >= 0) {
       sum += a[i];
    }
    return sum;
  }
}
```

Inheritance

- In JML, a subclass inherits specifications such as preconditions, postconditions, and invariants from its superclasses and interfaces that it implements.
- An interface also inherits specifications of the interfaces that it extends.

Extensions to Java expressions

- `\result` result of a method call
- `A ==> B` A implies B
- `A <==> B` A if and only if B
- `\old(E)` value of E in pre-state
- `\forall` and `\exists` - universal and existential quantifiers
  - `(\forall int i,j; 0 <= i && i < j && j < 10; a[i] < a[j])`
- `\max`, `\min`, `\product`, and `\sum`
  - ```
    (\sum int i; 0 <= i && i < 5; i) == 0 + 1 + 2 + 3 + 4
    (\product int i; 0 < i && i < 5; i) == 1 * 2 * 3 * 4
    (\max int i; 0 <= i && i < 5; i) == 4
    (\min int i; 0 <= i && i < 5; i-1) == -1
    ```
    ```

Extensions to Java expressions

- **`\num_of`**, returns the number of values for its variables for which the range and the expression in its body are true
  - `(\num_of T x; R(x); P(x)) == (\sum T x; R(x) && P(x); 1L)`
- Set comprehension
  - `new JMLObjectSet {Integer i | myIntSet.has(i) && i != null && 0 <= i.getInteger() && i.getInteger() <= 10 }`
- **`\duration(`*`mc`*`)`**, describes the specified maximum number of virtual machine cycle times to execute the method call
- **`\elemtype`**, which returns the most-specific static type shared by all elements of its array argument
  - `\elemtype(\type(int[]))` is `\type(int)`

Extensions to Java expressions

- **`\fresh`**, asserts that objects were freshly allocated (not allocated in the pre-state)
- **`\nonnullelements ==`**
  - **`myArray != null && (\forall int i; 0 <= i && i < myArray.length; myArray[i] != null)`**
- **`\typeof(E)`**, returns the most-specific dynamic type of an expression's value (null means unspecified)
- **`<:`**, compares two reference types
- **`\type`**, marks types in expressions.
  - **`\typeof(myObj) <: \type(PlusAccount)`**

Extensions to Java expressions

- `\invariant_for(o)`, true when its argument satisfies the invariant for its static type
  - `\invariant_for((MyObj)o)`
- `\is_initialized(o)`
- `\lockset`, set of locks held by current thread
- `\not_modified`, asserts that the values of objects are the same in pre- and poststates
- `\reach(x)`, the set of all objects accessible through x
- `\space(o)`, the amount of heap space allocated to o
- `\working_space(o.m(..))`, describes the maximum amount of heap space used by the method call

Example

```
public class Purse {
   final int MAX_BALANCE;
   int balance;
   //@ invariant 0 <= balance && balance <= MAX_BALANCE;

   byte[] pin;
   /*@ invariant pin != null && pin.length == 4
     @ && (\forall int i; 0 <= i && i < 4;
     @ 0 <= pin[i] && pin[i] <= 9);
     @*/

   /*@ requires amount >= 0;
     @ assignable balance;
     @ ensures balance == \old(balance) - amount
       @ && \result == balance;
     @ signals (PurseException) balance == \old(balance);
     @*/
```

Example cont.

```
   int debit(int amount) throws PurseException {
       if (amount <= balance) {
           balance -= amount; return balance;}
       else {
           throw new PurseException("overdrawn by" + amount);}
   }

   /*@ requires 0 < mb && 0 <= b && b <= mb
     @ && p != null && p.length == 4
     @ && (\forall int i; 0 <= i && i < 4;
     @ 0 <= p[i] && p[i] <= 9);
     @ assignable MAX_BALANCE, balance, pin;
     @ ensures MAX_BALANCE == mb && balance == b
     @ && (\forall int i; 0 <= i && i < 4; p[i]==pin[i]);
     @*/
   Purse(int mb, int b, byte[] p) {
      MAX_BALANCE = mb; balance = b; pin = (byte[])p.clone();
   }
}
```

JML tools

- Runtime assertions checking
  - JML compiler `jmlc`
- Testing
  - Jmlunit combines runtime assertion checking with unit testing
- Tools for generating specifications
  - Daikon infers likely invariants by observing runtime behavior of a program
  - Jmlspec can produce a skeleton of a specification file from Java source
- Documentation
  - Jmldoc produces browsable HTML from JML specifications

JML tools

- Static checking and verification
  - ESC/Java can automatically detect certain common errors and check relatively simple assertions.
  - JACK, similar to ESC/Java
  - LOOP, translates JML annotated code to PVS proof obligations
  - CHASE, checks some aspects of frame conditions

BCSL

- Motivation: bringing PCC to Java
- If one wants specify the behavior of bytecode, there has to be an assertion language for it
- BCSL, or Bytecode Specification Language, is meant as the low-level counterpart of JML
- Being designed in INRIA Sophia-Antipolis

- BCSL is a representative subset of JML, including
  - Class invariants, history constraints
  - Model/ghost variables
  - Method pre, post, exceptional conditions, frame conditions
  - Inner method specifications (loop invariants)
  - Expressions from Java (field access etc.)
  - Specification operators `\typed`, `\type`, `\elemtype`, `\old`, `\result`
- It includes the following features JML lacks:
  - Loop frame condition, which declares the locations that can be modified during a loop
  - Stack expressions `cntr` for stack counter and `st`(*AE*) standing for a stack element at position *AE*.

Compiling JML to BML

- Class files have an attribute table
  - It can have an unlimited number of attributes
  - A Java virtual machine implementation is required to silently attributes in the attributes table it doesn't recognize
  - So annotations can be included as extra attributes
- Java compilers generate Line Number Table and Local Variable Table attributes for class files
  - A JML compiler can take an existing classfile, and infer from the LNT and LVT how to associate the annotations with the class.

References

- http://www.cs.iastate.edu/~leavens/JML/ contains documentation, relevant papers and links to tools.