Applications of Formal Verification

Functional Verification of Java Programs: Java Modelling Language

Prof. Dr. Bernhard Beckert · Dr. Vladimir Klebanov | SS 2010
Design by Contract

Idea

Specifications fix a **contract** between caller and callee of a method (between client and implementor of a module):

- If caller guarantees precondition
- then callee guarantees certain outcome

- Interface documentation
- Contracts described in a mathematically precise language (JML)
  - higher degree of precision
  - *automation* of program analysis of various kinds
    (runtime assertion checking, **static verification**)
- Note: Errors in specifications are at least as common as errors in code,
Java comments with `@` as first character are JML specifications.
Modifiers to specification cases have no influence on their semantics.

- `public` specification items cannot refer to `private` fields.
- Private fields can be declared public for specification purposes only.

```java
class ATM {
    private /*@ spec_public @*/ BankCard insertedCard = null;
    private /*@ spec_public @*/
        boolean customerAuthenticated = false;

    /*@ public normal_behavior ... @*/
```
Method Contracts

/*@ requires r;
  @ assignable a;
  @ diverges d;
  @ ensures post;
  @ signals_only E1, ..., En;
  @ signals(E e) s;
  @*/
T m(...);

/*@ requires r;  // what is the caller’s obligation?
  @ assignable a;
  @ diverges d;
  @*/
T m(...);

Abbreviations

normal behavior = signals(Exception) false;

exceptional behavior = ensures false;

keyword 'also' separates the contracts of a method

Java Modelling Language

Prof. Dr. Bernhard Beckert · Dr. Vladimir Klebanov – Applications of Formal Verification

SS 2010 8/40
//@ invariant i;

- can be placed anywhere in a class (or interface)
- express global consistency properties (not specific to a particular method)
- must hold “always”
  (cf. visible state semantics, observed state semantics)
- instance invariants can, static invariants cannot refer to this
- default: instance within classes, static within interfaces
Pure Methods

Pure methods terminate and have no side effects.

After declaring

```java
public /*@ pure @*/ boolean cardIsInserted() {
    return insertedCard != null;
}
```

cardIsInserted()

could replace

```java
insertedCard != null
```

in JML annotations.
`pure` ≈ `diverges false;` + `assignable \ nothing;`
Expressions

- All Java expressions without side-effects
- $\Rightarrow$, $\Leftrightarrow$: implication, equivalence
- $\forall$, $\exists$
- $\text{num}\_of$, $\text{sum}$, $\text{product}$, $\text{min}$, $\text{max}$
- $\text{old}(\ldots)$: referring to pre-state in postconditions
- $\text{result}$: referring to return value in postconditions
Quantification in JML

\(\forall \text{int } i;\ 0\leq i && i<\text{result}.\text{length};\ \text{result}[i]>0\)

equivalent to

\(\forall \text{int } i;\ 0\leq i && i<\text{result}.\text{length} \implies \text{result}[i]>0\)

\(\exists \text{int } i;\ 0\leq i && i<\text{result}.\text{length};\ \text{result}[i]>0\)

equivalent to

\(\exists \text{int } i;\ 0\leq i && i<\text{result}.\text{length} \land \text{result}[i]>0\)

- Note that quantifiers bind two expressions, the range predicate and the body expression.
- A missing range predicate is by default true.
- JML excludes null from the range of quantification.
Generalised and Numerical Quantifiers

\(\text{\texttt{\num_of C c; e}}\)

\(\#\{c|e\}\), number of elements of class \(C\) with property \(e\)

\(\text{\texttt{\sum \mathit{C} c; p; t}}\)

\(\sum_{c:[p]} [t]\)

\(\text{\texttt{\product \mathit{C} c; p; t}}\)

\(\prod_{c:[p]} [t]\)

\(\text{\texttt{\min \mathit{C} c; p; t}}\)

\(\min_{c:[p]} \{[t]\}\)

\(\text{\texttt{\max \mathit{C} c; p; t}}\)

\(\max_{c:[p]} \{[t]\}\)
The assignable Clauses

Comma-separated list of:

- `e.f` (where `f` a field)
- `a[*]`, `a[x..y]` (where `a` an array expression)
- `\nothing`, `\everything` (default)

Example

```java
C x, y;
//@ assignable x, x.i;
void m() {
    C tmp = x;  //allowed (local variable)
    tmp.i = 27; //allowed (in assignable clause)
    x = y;      //allowed (in assignable clause)
    x.i = 27;   //forbidden (not local, not in assignable)
}
```
The `diverges` Clause

`diverges e;`

with a boolean JML expression `e` specifies that the method may not terminate only when `e` is true in the pre-state.

**Examples**

`diverges false;`

The method must always terminate.

`diverges true;`

The method may terminate or not.

`diverges n == 0;`

The method must terminate, when called in a state with `n != 0`.
The signals Clauses

```java
ensures p;
signals_only ET1, ..., ETm;
signals (E1 e1) s1;
...
signals (En en) sn;
```

- normal termination  $\Rightarrow$ $p$ must hold (in post-state)
- exception thrown  $\Rightarrow$ must be of type $ET1, \ldots, ETm$
- exception of type $E1$ thrown  $\Rightarrow$ $s1$ must hold (in post-state)
  ...
- exception of type $En$ thrown  $\Rightarrow$ $sn$ must hold (in post-state)
public interface IBonusCard {

    public void addBonus(int newBonusPoints);

}

public interface IBonusCard {

    /*@
    public instance model int bonusPoints; @*/

    public void addBonus(int newBonusPoints);

}

public interface IBonusCard {

    /*@
    public instance model int bonusPoints; @*/
    /*@
    ensures bonusPoints == \old(bonusPoints) + newBonusPoints; @*/

    public void addBonus(int newBonusPoints);

}

How to add contracts to abstract methods in interfaces?
Remember: There are no attributes in interfaces.
More precisely: Only static final fields.
Implementing Interfaces

```java
public interface IBonusCard {
    /*@ public instance model int bonusPoints; @*/

    /*@ ... @*/

    public void addBonus(int newBonusPoints);
}
```

Implementation

```java
public class BankCard implements IBonusCard{
    public int bankCardPoints;

    /*@ private represents bonusPoints = bankCardPoints; @*/

    public void addBonus(int newBonusPoints) {
        bankCardPoints+=newBonusPoints;
    }
}
```
Other Representations

```java
/*@ private represents bonusPoints
    = bankCardPoints; */

/*@ private represents bonusPoints
    = bankCardPoints * 100; */

/*@ represents x \such_that A(x); */
```
An invariant to a class is inherited by all its subclasses.

An operation contract is inherited by all overridden methods.
It can be extended there.
Other JML Features

- assertions ‘//@ assert e;’
- loop invariants ‘//@ maintaining p;’
- data groups
- refines
- many more…
Nullity

JML has modifiers `nonnull` and `nullable`

private /*@spec_public non_null@*/ Object x;

\[\Rightarrow\] implicit invariant added to class: ‘invariant x != null;’

void m(/*@non_null@*/ Object p);

\[\Rightarrow\] implicit precondition added to all contracts:
‘requires p != null;’

/*@non_null@*/ Object m();

\[\Rightarrow\] implicit postcondition added to all contracts:
‘ensures \result != null;’

`nonnull` is the default!
If something may be null, you have to declare it `nullable`
Problems with Specifications Using Integers

/*@ requires y >= 0; @
@ ensures @
@ \result \times \result <= y && @
@ y < (abs(\result)+1) \times (abs(\result)+1); @
@ */

public static int isqrt(int y)

For \( y = 1 \) and \( \result = 1073741821 = \frac{1}{2}(\text{max}\_\text{int} - 5) \) the above postcondition is true, though we do not want 1073741821 to be a square root of 1.

JML uses the Javasemantics of integers:

\[
\begin{align*}
1073741821 \times 1073741821 &= -2147483639 \\
1073741822 \times 1073741822 &= 4
\end{align*}
\]
JML Tools

Many tools support JML (see JML homepage). Among them:

- **jml**: JML syntax checker
- **jmldoc**: code documentation (like Javadoc)
- **jmlc**: compiles Java+JML into bytecode with assertion checks
- **jmlunit**: unit testing (like JUnit)
- **rac**: runtime assertion checker
- **ESC/Java2**: lightweight static verification
- **KeY**: full static verification
- **OpenJML**: tool suite, under development

The tools do not yet support the new features of Java 5! e.g.: no generics, no enums, no enhanced for-loops, no autoboxing