Applications of Formal Verification

Functional Verification of Java Programs:
Java Modeling Language

Dr. Vladimir Klebanov · Dr. Mattias Ulbrich · (Folien nach Prof. Dr. Bernhard Beckert) | SS 2015
"BISL"

short for

**Behavioural Interface Specification Language**

- used to describe formally input/output-behaviour of operations
- abstraction from implementation details
  - code structure, algorithms and data structures
- tailored for a particular programming language

**Example BISLs:**

<table>
<thead>
<tr>
<th>BISL</th>
<th>For</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Modeling Language Spec#</td>
<td>Java</td>
</tr>
<tr>
<td>ACSL</td>
<td>C#</td>
</tr>
<tr>
<td>VCC</td>
<td>C (tool: Frama-C)</td>
</tr>
<tr>
<td>LARCH/C/C++</td>
<td>C/C++ (discontinued)</td>
</tr>
<tr>
<td>JAZZ</td>
<td>Java (discontinued)</td>
</tr>
</tbody>
</table>
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:
  “Behavioural Interface Specification Language”

- 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
Within a JML annotation, an ‘@’ is ignored:
Visibility Modifiers

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
public class ATM {
    private @spec_public BankCard insertedCard = null;
    private @spec_public boolean customerAuthenticated = false;

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

T m(...);

/*@ requires r; // what is the caller’s obligation?
 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;
 ensures post;
 signals_only E1,...,En;
 signals(E e) s;
@*/

T m(...);

/*@ requires r; // what is the caller’s obligation?
 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;
 ensures post;
 signals_only E1,...,En;
 signals(E e) s;
@*/

T m(...);

/*@ requires r; // what is the caller’s obligation?
 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;
 ensures post;
 signals_only E1,...,En;
 signals(E e) s;
@*/

T m(...);
Class Invariants

//@ 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, ownership, dynamic frames)
- 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.

Hence, they can be used in JML specifications.

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
- \(\forall\), \(\exists\)
- \(\text{num} \_\text{of}, \sum, \text{product}, \min, \max\)
- \(\text{old}(...)\): referring to pre-state in postconditions
- \(\text{result}\): referring to return value in postconditions
Quantification in JML

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

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

- 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

\( \text{\texttt{num\_of \, T \, i; \, e}} \)

\( \#\{i|e\} \), number of elements of type \( T \) with property \( e \)

\( \text{\texttt{sum \, T \, i; \, p; \, t}} \)

\( \sum [t] \)

\( \text{\texttt{product \, T \, i; \, p; \, t}} \)

\( \prod [t] \)

\( \text{\texttt{min \, T \, i; \, p; \, t}} \)

\( \text{\texttt{max \, T \, i; \, p; \, t}} \)

\( \min \{[t]\} \)

\( \max \{[t]\} \)
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

C x, y; int i;
//@ 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)
}
“Nothing” is more than you think it is

assignable \nothing means that no memory location existing at method invocation must be changed.

Valid specification

```java
//@ assignable \nothing;
void n() {
    C c = new C();
    c.i = 42;
}
```

```java
//@ assignable \nothing;
void n() {
    C c = new C();
    c.i = 42;  // allowed: fresh objects can be modified
}
```
The diverges Clause

\begin{verbatim}
  diverges e;
\end{verbatim}

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

**Examples**

\begin{verbatim}
  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 \( n \neq 0 \).
\end{verbatim}
The signals Clauses

\[
\begin{align*}
\text{ensures } & \ p; \\
\text{signals\_only } & \ ET_1, \ldots, \ ET_m; \\
\text{signals } & \ (E_1 \ e_1) \ s_1; \\
\ldots \\
\text{signals } & \ (E_n \ e_n) \ s_n;
\end{align*}
\]

- normal termination  \( \Rightarrow \)  \( p \) must hold (in post-state)
- exception thrown  \( \Rightarrow \)  must be of type \( ET_1, \ldots, \) or \( ET_m \)
- exception of type \( E_1 \) thrown  \( \Rightarrow \)  \( s_1 \) must hold (in post-state)
  \( \ldots \)
- exception of type \( E_n \) thrown  \( \Rightarrow \)  \( s_n \) must hold (in post-state)
Model Fields

```java
public interface IBonusCard {

    public void addBonus(int newBonusPoints);

}
```

```java
public interface IBonusCard {

    /*@ public instance model int bonusPoints; */

    public void addBonus(int newBonusPoints);

}
```

```java
public interface IBonusCard {

    /*@ assignable bonusPoints; */

    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; @*/
```

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

```java
/*@ represents x \such that A(x); @*/
```
Inheritance of Specifications in JML

Behavioral Subtyping, (Lizkov-Leavens Substitution Principle)

If $D$ is a subclass of $C$, then objects of type $C$ may be replaced with objects of type $D$ without altering the desirable properties of the program.

- A class invariant is inherited by all subclasses.
- An operation contract is inherited by all overridden methods.
- Subclass may add invariants and contracts
Dealing with loops

- Loops are a challenge for reasoning about programs
- Loop specifications to guide program proof systems

```markdown
/*@ loop_invariant linv;
@ decreases variant;
@ assignable A;
@*/

while(...) { ... }
```

- Loop invariant `linv` needs to hold for all iterations (every time the loop condition is checked)
- The `variant` must be decreasing non-negative integer (termination – there is no infinite decreasing sequence)
- assignable, *cf. assignable for methods*
Other JML Features

- assertions `//@ assert e;`
- assumptions `//@ assume e;`
- data groups
- refines
- many more...
Nullity

JML has modifiers `non_null` and `nullable`

```java
private /*@spec_public non_null@*/ Object x;
⇝ implicit invariant added to class: `invariant x != null;`
```

```java
void m(/*@non_null@*/ Object p);
⇝ implicit precondition added to all contracts:
`requires p != null;`
```

```java
/*@non_null@*/ Object m();
⇝ implicit postcondition added to all contracts:
`ensures \result != null;`
```

`non_null` is the default!

If something may be `null`, you have to declare it `nullable`
Problems with Specifications Using Integers

/*
 @ requires \( y \geq 0 \);
 @ ensures \( \text{result} \geq 0 \);
 @ ensures \( \text{result} \times (\text{result} + 1) > y \);
 @ ensures \((\text{result} + 1) \times (\text{result} + 1) > y \);
 @
 */

public static int isqrt(int y)

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

JML uses the Java semantics of integers:

\[ 1073741821 \times 1073741821 = 2147483639 \]

The JML type \( \text{bigint} \) provides arbitrary precision integers.

Klebanov, Ulbrich – Applications of Formal Verification
JML Tools

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

- **KeY**: full static verification
- **OpenJML**: tool suite, under development
- **jml**: JML syntax checker
- **jmldoc**: code documentation (like Javadoc)
- **jmlunit**: unit testing (like JUnit)
- **JMLUnitNG**: unit test generation
- **ESC/Java2**: lightweight static verification

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