Verificare, Validare şi Testare Automată

PPPFP Informatică

2019-2020/ I

Java Modeling Language (JML)

Outline

- Introduction
  - Motivation
  - Formal Specification Languages
  - Design by Contract
- JML
  - What is JML?
  - Background
  - JML Annotations
  - JML Specification Expressions
  - DbC and JML
  - Notes on JML
- JML Tools
  - jmlc/jmlrac and ESC2Java
  - Demos

- Motivation
- Formal Specification Languages
- Design by Contract

INTRODUCTION

Motivation

- Why should we use formal methods?
  - civil engineering case
    - assuming that we want to build a house...

    - How can we make sure that the built house will not collapse?

- Things to do:
  1) make an (abstract) model
  2) specify properties for the model
  3) verify/check the properties

Motivation (2)

- Why should we use formal methods?
  - software engineering case
    - assuming that we want to write a software…

```
public interface HouseUtility{
        public void openWindow();
        public void closeWindow();
        //...
}
```

- How can we make sure that the built software works correctly?

- Things to do:
  1) make a ***formal model***
  2) ***specify properties*** for the model
  3) ***verify/check*** the properties

Formal Methods in Software Engineering

- Existing formal approaches:

  1) ***modeling languages:*** finite state machines, Z, guarded command language, UML, …

  2) ***languages for specifying properties:*** predicate logic, Hoare logic, JML, …

  3) ***techniques for checking*** the specified properties: testing, model checking, theorem proving, …

- Formal languages guarantee:
  - Precision (no ambiguity);
  - Certainty (modeling errors);
  - Automation (automatic verification tools).

Formal Methods in Software Engineering (2)

- Formal methods and JML:
  1) formal model is ***Java programming language***
  2) the properties are specified in ***JML***
  3) Properties may be
     - ***Tested*** using ***jmlrac***
     - ***Checked*** using ***ESC2Java***

Design by Contract

- Design by Contract (DbC)
  - a general design paradigm
  - states that the first elements of code written for a method, class, or program should be its contracts
- DbC was made popular by the Eiffel programming language, which has contracts built-in
- useful for
  - enabling better compile-time checks
  - enabling easier run-time checks (aspects fit perfectly here)
  - self-documenting code
  - assigning “blame”
- in DbC, contracts are a critical step between understanding what a program should do and implementing a program that does it:
  - once the programmer understands what a section of code should do, he can write contracts for it
  - once contracts have been written, it is more clear what the actual implementation should be

Design by Contract (2)

- Pre- and post-conditions define a **contract** between a class and its clients:
  - Client must:
    - **ensures** precondition
    - **requires** post-condition;
  - Method (software provider) should/must:
    - **requires** precondition;
    - **ensures** post-condition.

What is JML?
Background
JML Annotations
JML Specification Expressions
DbC and JML
Notes on JML

JML

What is JML?

- Java Modeling Language (JML)
  - ***formal specification language* for Java**
  - **allows:**
    - **to specify behavior of Java classes**
    - **to record design/implementation decisions**
  - **by adding *annotations* to Java source code (e.g., *preconditions, postconditions, class invariants*)**

Background

- JML started in 1997 in an effort, led by Gary T. Leavens, to create formal, reusable specification language for Java.
- JML has seen grown in popularity and is used as the specification language for many different tools and research projects, including
  - Runtime assertion checkers (e.g. **jmlc/jmlrac**)
  - Test generation (e.g. jmlunit)
  - Static checkers (**ESC2Java**)
  - Formal verification tools (e.g. KeY)
  - Design tools (e.g. AutoJML)
- Industry includes verification specification on security-critical applications such as:
  - financial data cards;
  - voting systems.

Pre- and Post-conditions

- method precondition
  - says what must be true before call it;
  - example: `//@ requires x >= 0;`
- method normal postcondition
  - says what is true when it returns normally (i.e., without throwing an exception);
  - example: `//@ ensures \result >= 0;`
- method exceptional postcondition
  - says what is true when a method throws an exception;
  - example: `//@ signals (IllegalArgumentException e) x < 0;`

Invariants

- (class) invariants
  - are properties that must be maintained by all methods;
  - are implicitly included in all pre- and post-conditions;
  - must also be preserved if an exception is thrown!
  - Example:

```
public class Account{
   public static final short MAX_VALUE = 1000;
   private short amount;
   /*@ invariant 0 <= amount &&
     @ amount <= MAX_VALUE;
     @*/
   //...
}
```

  - Advantages:
    - document design decisions;
    - help in understanding the code.

Assertions

- assert clause
  - specifies a property that should hold at some point in the code;
  - example:

```
if (i <= 0 || j < 0) {
   //...
} else if (j < 5) {
   //@ assert i > 0 && 0 < j && j < 5;
} else {
   //@ assert i > 0 && j > 5;
   //...
}
```

- JML keyword `assert` introduced in Java (since Java 1.4).

Pure Methods

- pure method
  - a method without side-effects;
  - example:

```
public /*@ pure @*/ int getAmount(){
    return amount;
}
```

- only pure methods, can be used in specifications
  - example:

```
//@ invariant 0<=getAmount() && getAmount()<=MAX_VALUE;
```

Common Pragmas - Summary

Pragma Syntax	Place to Appear	Semantics
`requires E`	Before a method	*E* must hold whenever the method is invoked
`ensures E`	Before a method	*E* must hold after the method terminates normally
`signals (T t) E`	Before a method	*E* must hold after the method terminates with exception of type *T*
`invariant E`	Class body	*E* must hold whenever no method of that class is running
`non_null`	Variable	Variable must never contain *null*
`spec_public`	Field	Field is considered public as far as specifications are concerned

- *E* – boolean *specification expression*

JML Annotations

- JML Annotation
  - a Java comment line starting with `@`;
  - may be composed of semicolon-separated pragmas:
  - examples:
    - `//@ This is an JML annotation;`
    - ```
      /*@ This is the first pragma
          @ in the current annotation;
          @ This is the second pragma
          @*/
      ```
  - *at*-sign (`@`) on the beginning of lines is ignored within Java Annotations;
  - `@` must be right next to the start of comment characters.
      ```

JML Comments

- JML Comment
  - syntax: `(* JML comment *)`
  - is actually considered an expression that returns `true`;
  - examples:
    - `//@ requires (* x is positive *);`
    - ```
      /*@ ensures (* \result is an approximation to
        @                    the square root of x *)
        @                && \result >= 0;
        @*/
      ```
- The easiest way to *comment out JML*?
  - solution: ***comment out the comment****, i.e., insert a <space> within the annotation****;***
  - example:
    - `//␣@ requires x >= 0.0;`
    - ```
      /*␣@ ensures  0 <= amount &&
        @          amount <= MAX_VALUE;
        @*/
      ```

JML Specification Expression

- **JML specification expression**
  - similar to a Java expression, with some differences that include:
    - no use of : assignment (=), object creation (`new`), increment `(++)`, decrement `(--)`, or method invocation (`m(a,b)`);
    - calls of “pure” methods are allowed;
    - special variables, operators, and loops;
    - special functions.

Syntax	Semantics
`a ==> b`	`a` implies `b`
`a <== b`	`b` implies `a`
`a <==> b`	`a iff b`
`a <=!=> b`	`! ( a <==> b )`

JML Special Specification Expressions

Syntax	Semantics
`\result`	the value returned from the method
`\old(A)`	The value of `A` as it was before the method was invoked
`(\forall T V; E)`	Declares all variables in `V` to be of type `T`, then verified `E` holds for every valid value for every variable
`(\exists T V; E)`	As `\forall`, only a single variable for which `E` will hold is enough

- `E` – boolean specification expression
- `A` – any specification expression
- `T` – type expression
- `V` – comma-separated list of variable names

DbC and JML

- DbC is emphasized within JML
  - example:

```
//@ requires x >= 0.0;
/*@ ensures JMLDouble.approximatelyEqualTo(x,
  @          \result * \result, eps);
  @*/
double sqrt(double x){...}
```

	Rights	Obligations
Client	to get a square root approximation	to pass a non-negative value
Implementation/ Provider	to assume/**require** the argument is a non-negative value	to compute/**ensure** the returned values is square root

Notes on JML

- JML syntax is not very well organized
  - Multiple keywords with the same meaning, inconsistent assertion syntax, etc.
- Why not use Java annotations?
  - This is a classical use-case for annotations!
  - Unfortunately, JML started before Java 1.6, and many tools already exist to support that form, so no conversion very soon.
- The programmer still has to figure out what specifications are needed.

Many Tools, One Language

`jmlc/jmlrac and Esc2Java`

Runtime Assertion Checker and Static Checker

JML TOOLS

Many Tools, One Language

- There are several types of tools that work on JML:
  - **JML compiler: jmlc;**
  - **static checker: ESC2Java;**
  - **run-time checker: jmlrac;**
  - unit tester: jmlunit;
  - model testing: BOGOR;
  - web pages: jmldoc;
  - correctness proof: JACK, Jive, KeY, LOOP, Krakatoa;
  - data trace file: Daikon;
  - etc.

`jmlc` / `jmlrac` and `Esc2Java`
JML Compiler and Runtime Assertion Checker

default (no specification checking)
javac & java usage

- class `Account.java`
  - compile:
    - `javac Account.java`
    - output: a bytecode file `Account.class`
    - ignores any comments, i.e., JML specification
  - run with the standard VM:
    - `java Account`
    - possible specification inconsistencies not highlighted

specification checking
jmlc & jmlrac usage

- class `Account.java`
  - compile:
    - `jmlc Account.java` or `jmlc -Q Account.java`
    - **output:** a bytecode file `Account.class` that enables automatic checks of JML assertions at the run time
    - **`jmlc` acts as a preprocessor for `javac`**
  - run the **JML** run-time **a**ssertion **c**hecker:
    - `jmlrac Account`
    - possible specification inconsistencies identified and `Errors` are thrown
    - `jmlrac` script enables the automatic use of JML runtime classes (`jmlruntime.jar`) from the Java boot class path, required to run the checks on the assertions
    - **`jmlrac` acts as a wrapper for the standard VM**

jmlc/jmlrac and Esc2Java
Runtime Assertion Checker and Static Checker

compile-time checking

Esc2Java

- **checks specifications at compile-time**
- proves the specification correctness
- warns about likely runtime exceptions and violations.
- automatically tries to prove simple JML assertions at compile time

run-time checking

jmlc & jmlrac

- **checks specifications at run-time**
- tests the specification correctness only
- finds the specification violations at runtime
- **jmlc** = special compiler that inserts runtime tests for all JML assertions;
- **jmlrac** - any assertion violation results in a special exception at run-time.

Run-time vs. Compile-time (static) Checking

- ESC2Java may miss an error that is actually present ;
- ESC2Java may warn of errors that are impossible;
- finds potential bugs quickly ;
- good at proving absence of runtime exceptions (e.g., Null-, ArrayIndex-OutOfBounds-, ClassCast-) and verifying relatively simple properties;
- ESC2Java independent of any test suite;
- results of runtime testing are only as good as the test suite;
- ESC2Java provides higher degree of confidence, for all pre- and post-conditions of methods and invariants.

Tool Installation

compile-time checking

Esc2Java

- download:
  - http://www.cs.virginia.edu/cs201j/escjava.html ;
  - http://www.kindsoftware.com/products/opensource/ESCJava2/download.html ;
- as Eclipse plug-in
  - http://kindsoftware.com/products/opensource/Mobius/updates/
- tutorials:
  - http://kindsoftware.com/products/opensource/ESCJava2/
  - https://piazza.com/class_profile/get_resource/h6i0bbjqlhy5wh/h7ft2ccr1jr4k2

run-time checking

jmlc & jmlrac

- download:
  - www.jmlspecs.org ;
  - http://www.openjml.org/documentation/installation.shtml
- as Eclipse plug-in
  - http://jmlspecs.sourceforge.net/openjml-updatesite
- JML installation notes:
  - http://www.cse.chalmers.se/edu/year/2012/course/_courses_2011/TDA566/JML-Installation-Notes.html
- tutorials:
  - http://www.cs.ru.nl/E.Poll/talks/jmlrac.pdf

Demos

compile-time checking

Esc2Java

- **Factorial.zip**
- **Account.zip**

run-time checking

jmlc & jmlrac

- **FastExponentiation.zip**
- **Bag.zip**