Formale Grundlagen der Informatik 3

The Java Modeling Language, Part I

Reiner Hähnle

17 December 2013

Road-map

First half of the course:

Modelling and verification of concurrent/distributed systems

Second half of course:

Deductive Verification of JAVA source code

1. Formally specifying JAVA programs
2. Proving JAVA programs correct

Specification = Precision

Economist:
The cows in Scotland are brown

Logician:
No, there are cows in Scotland of which one at least is brown!

Computer Scientist:
No, there is at least one cow in Scotland, which on one side is brown!!

Specification Target

System level specification
(requirements analysis, GUI, use cases)
important, but
not subject of this course

We focus on:

Unit specification—contracts between implementors on various levels:

- Application level $\leftrightarrow$ application level
- Application level $\leftrightarrow$ library level
- Library level $\leftrightarrow$ library level

Unit Specifications (“Komponentenspezifikationen”)

Cf. unit testing (“Modultest”, “Komponententest”)

In Object-Oriented Setting:

Units to be specified are interfaces, classes, and their methods

First focus on methods

Method specifications must comprise the following aspects:

- Result value
- Initial values of formal parameters
- Accessible part of pre-/post-state

Specifications as Contracts

Useful analogy to stress the different roles/obligations/responsibilities in a specification:

Specification as a contract (between method implementor and user)

"Design by Contract" methodology (Meyer, 1992, EIFFEL)

Contract between caller and callee (called method)

Callee guarantees certain outcome provided caller guarantees prerequisites

Specifications as Contracts: Example

"Wenn Sie die Ente hereinlassen, lasse ich das Wasser heraus!"

Running Example: ATM.java

```
public class ATM {

    // fields:
    private BankCard insertedCard  = null;
    private int wrongPINCounter  = 0;
    private boolean customerAuthenticated  = false;

    // methods:
    public void insertCard (BankCard card) { ... }
    public void enterPIN (int pin)  { ... }
    public int accountBalance () { ... }
    public int withdraw (int amount) { ... }
    public void ejectCard () { ... }
}
```

Informal Specification

Very informal specification of `enterPIN (int pin)`

> *"Enter the PIN that belongs to the currently inserted bank card into the ATM. If a wrong PIN is entered three times in a row, the card is confiscated. After having entered the correct PIN, the customer is regarded as authenticated."*

Becoming More Precise: Specification as Contract

Contract states what is guaranteed under which conditions

precondition	card is inserted, user not yet authenticated, card is valid, PIN is correct
postcondition	user is authenticated

precondition	card is inserted, user not yet authenticated, `wrongPINCounter < 2`, PIN is incorrect
postcondition	`wrongPINCounter` is increased by 1 user is not authenticated

precondition	card is inserted, user not yet authenticated, `wrongPINCounter >= 2`, PIN is incorrect
postcondition	card is confiscated, card is made invalid user is not authenticated

Meaning of Pre-/Post-Condition Pairs

Definition

A **pre-/post-condition** pair for a method `m` is **satisfied by the implementation** of `m` if:

> *When* `m` *is called in any state that satisfies the precondition then in any terminating state of* `m` *the postcondition is true.*

Remarks

1. No guarantee when the precondition is not satisfied
2. Termination may or may not be guaranteed
3. Terminating state may be reached by normal or by abrupt termination (e.g., exception)

Formal Specification

Natural language specs are very important and widely used

This course's focus is

Formal Specification

Describe contracts of units with mathematical rigour

Motivation

- High degree of precision
  - formalization often exhibits omissions/inconsistencies
  - avoid ambiguities inherent to natural language
- Potential for automation of program analysis
  - run-time assertion checking
  - test case generation
  - program verification

Java Modeling Language (JML)

JML is a specification language tailored to JAVA

General JML Philosophy

Integrate

- specification
- implementation

in one single language

$\Rightarrow$ JML is not external to JAVA, but an extension of JAVA

JML
is
JAVA + FO Logic + pre-/post-conditions, invariants + more ...

JML Annotations

JML extends Java by annotations

JML annotations include:

- ✔ preconditions
- ✔ postconditions
- ✔ class invariants
- ✔ additional modifiers
- ✘ "specification-only" fields
- ✘ "specification-only" methods
- ✔ loop invariants
- ✔ ...
- ✘ ...

✔: in this course, ✘: not in this course

JML/Java integration

JML annotations are attached to Java programs
by
writing them directly into the Java source code files

Ensures compatibility with standard Java compiler:

JML annotations live in special Java comments,
ignored by Java compiler, recognized by JML tools

JML as JAVA Comments

From the file ATM.java

⋮

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) { ... }
```

⋮

Everything between /* and */ is invisible for JAVA compiler

JML as JAVA Comments

JAVA comment lines starting with @ read and parsed by JML tools

```
/*@ public normal_behavior
  @ requires !customerAuthenticated; @ only to beautify
    requires pin == insertedCard.correctPIN;
  @*/
//@ ensures customerAuthenticated; rest-of-line comment
//␣@ ensures !customerAuthenticated; no JML: @ not first
public void enterPIN (int pin) { ... }
```

JML by Example: Public Modifier

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) { ... }
```

This is a **`public`** specification case:

1. it is accessible from all classes and interfaces
2. it can only refer to `public` fields/methods of this class (can be problematic, come back to it later)

In this course: mostly **`public`** specifications

JML by Example: Specification Cases

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) { ... }
```

Each keyword ending with **`behavior`** opens a specification case

`normal_behavior` Specification Case

The called method guarantees to not throw an exception,
if the caller guarantees all preconditions of this specification case

JML by Example: Preconditions

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) { ... }
```

Specification case has two preconditions (marked by **requires**)

Here:
preconditions happen to be boolean JAVA expressions

In general:
preconditions are boolean JML expressions (including quantifiers)

JML by Example: Preconditions Cont'd

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
```

Both preconditions must be true in prestate

Equivalent to:

```
/*@ public normal_behavior
  @ requires (    !customerAuthenticated
  @              && pin == insertedCard.correctPIN );
  @ ensures customerAuthenticated;
  @*/
```

JML by Example: Postconditions

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) { ... }
```

Specification case has one postcondition (marked by **`ensures`**)

- Postconditions are boolean JML expressions
- If there is more than one **`ensures`** clause:
  postcondition is the conjunction of all clauses

JML by Example

Multiple specification cases connected by **also**

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @
  @ also
  @
  @ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin != insertedCard.correctPIN;
  @ requires wrongPINCounter < 2;
  @ ensures wrongPINCounter == \old(wrongPINCounter) + 1;
  @*/
public void enterPIN (int pin) { ... }
```

JML by Example: Access of Prestate

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin != insertedCard.correctPIN;
  @ requires wrongPINCounter < 2;
  @ ensures wrongPINCounter == \old(wrongPINCounter) + 1;
  @*/
public void enterPIN (int pin) { ...
```

Access to value of prestate in postcondition

`\old(`*E*`)` means: *E* evaluated in the prestate (of `enterPIN()`)

- `\old(`*E*`)` is a JML expression that is not a JAVA expression
- *E* can be any (arbitrarily complex) JAVA/JML expression

Specification Cases Complete?

```
@ public normal_behavior
@ requires !customerAuthenticated;
@ requires pin == insertedCard.correctPIN;
@ ensures customerAuthenticated;
```

What does specification case not tell about poststate?

Fields of class `ATM`:

```
insertedCard
customerAuthenticated
wrongPINCounter
```

What happens with `insertedCard` and `wrongPINCounter`?

Completing Specification Cases

```
@ public normal_behavior
@ requires !customerAuthenticated;
@ requires pin == insertedCard.correctPIN;
@ ensures customerAuthenticated;
@ ensures insertedCard == \old(insertedCard);
@ ensures wrongPINCounter == \old(wrongPINCounter);
```

- Similar postconditions added for the other specification cases
- Assumption that environment is unchanged unless explicitly stated: usually called frame condition

Clearly unsatisfactory to add

```
@ ensures loc == \old(loc);
```

for all locations loc which do not change

Assignable Locations

More efficient to explicitly list all locations that may change:

`@ assignable` loc_1`, . . . ,`loc_n`;`

Assignable clause: value of no location besides $loc_1, \ldots, loc_n$ can change (but could change temporarily during execution of method)

Special cases of assignable clause

No location may be changed:

```
@ assignable \nothing;
```

Unrestricted, method allowed to change anything:

```
@ assignable \everything;
```

This is the default if no assignable clause is given

Specification Case with Assignable

```
@ public normal_behavior
@ requires insertedCard != null;
@ requires !customerAuthenticated;
@ requires pin != insertedCard.correctPIN;
@ requires wrongPINCounter >= 2;
@ ensures insertedCard == null;
@ ensures \old(insertedCard).invalid;
@ assignable wrongPINCounter,
@            insertedCard,
@            insertedCard.invalid;
```

JML Modifiers

JML extends the JAVA modifiers by additional modifiers

The most important ones are:

- **`spec_public`**
- **`pure`**

JML Modifiers: `spec_public`

In "`enterPIN`" example, pre-/postconditions made use of class fields

But: **`public`** specifications can access only **`public`** fields

Not desired: make all fields mentioned in specification public

Control visibility with `spec_public`

- Keep visibility of JAVA fields **`private/protected`**
- If necessary make them visible in specification only by **`spec_public`**

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

(different solution, not discussed here: use specification-only fields)

JML Modifiers: `pure`

Specifications more concise with method calls inside JML annotations

Example

- `o1.equals(o2)`
- `li.contains(elem)`
- `li1.max() < li2.min()`

Specifications may not themselves change the state!

Definition (Pure method)

A JAVA method is called pure iff it has no side effects and it always terminates. Specifically, it may create no new objects.

JML expressions may call pure methods. These are annotated by **`pure`**

```
public /*@ pure @*/ int max() { ... }
```

JML Modifiers: `pure` Cont'd

How do we know that a **`pure`** method is really pure?

- **`pure`** puts obligation on implementor not to cause side effects
- It is possible to formally verify that a method is pure
  - Write a contract that expresses purity and verify it
- **`pure`** implies **`assignable \nothing`**;
- Assignable clauses can be local to a specification case while **`pure`** fixes behavior of a method

JML Expressions $\neq$ Java Expressions

Definition (JML Expressions—to be completed)

- Each side-effect free Java expression is a JML expression
  - Any method call must be to pure method
  - E.g., `i++` is not a JML expression
- If E is a side-effect free Java expression, then `\old(E)` is a JML expression
- If `a` and `b` are **`boolean`** JML expressions then
  - `!a` ("not a")
  - `a && b` ("a and b")
  - `a || b` ("a or b")
  - `a ==> b` ("a implies b")
  - `a <==> b` ("a is equivalent to b")

  are also **`boolean`** JML expressions.

But this is not enough!

Beyond `boolean` Java expressions

How to express the following?

- An array "**int** a" contains only non-negative elements
- The variable `m` holds a maximal element of array `a`
- All `Account` objects in the array `accountProxies` are stored at the index corresponding to their respective `accountNumber` field
- All created instances of class `BankCard` have different `cardNumbers`

Quantified JML Expressions

Definition (JML Expressions)

- Each side-effect free Java expression is a JML expression
- If E is a side-effect free Java expression, then `\old(E)` is a JML expression
- If `a` and `b` are **`boolean`** JML expressions, `x` is a variable of type `t`:
  - `!a` ("not a")
  - `a && b` ("a and b")
  - `a || b` ("a or b")
  - `a ==> b` ("a implies b")
  - `a <==> b` ("a is equivalent to b")
  - `(\forall t x; a)` ("for all `x` of type `t`, `a` is true")
  - `(\exists t x; a)` ("there exists `x` of type `t` such that `a`")
  - `(\forall t x; a; b)` ("for all `x` of type `t` fulfilling `a`, `b` is true")
  - `(\exists t x; a; b)` ("there exists an `x` of type `t` fulfilling `a`, such that `b` is true")

  are also **`boolean`** JML expressions.

Range Predicates

JML quantifiers (optionally) have more general syntax than FOL ones

Definition (Range predicate)

In the JML expressions `(\forall t x; a; b)` and `(\exists t x; a; b)` the **`boolean`** a is called range predicate.

Range predicates are syntactic sugar for standard FOL quantifiers:

`(\forall t x; a; b)`
equivalent to
`(\forall t x; a ==> b)`

`(\exists t x; a; b)`
equivalent to
`(\exists t x; a && b)`

Pragmatics of Range Predicates

Range predicates used to restrict range of `x` further than to its type `t`

Example

"Array `a` is sorted between indices 0 and 9":

```
(\forall int i,j; 0<=i && i<j && j<10; a[i] <= a[j])
```

Using Quantified JML Expressions

- An array int a contains only non-negative elements

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

- The variable m holds a maximal element of array a

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

  Is this sufficient? Need in addition:

  `(\exists int i; 0 <= i && i < a.length; m == a[i])`

Using Quantified JML Expressions Cont'd

- All `Account` objects in the array `accountProxies` are stored at the index corresponding to their respective `accountNumber` field

```
(\forall int i; 0 <= i && i < maxAccountNumber;
              accountProxies[i].accountNumber == i )
```

- All created instances of class `BankCard` have different `cardNumbers`

```
(\forall BankCard b1, b2;
         \created(b1) && \created(b2);
         b1 != b2 ==> b1.cardNumber != b2.cardNumber)
```

Remarks

- Restrict range to created objects with JML keyword `\created`
- JML/KeY quantifiers range even over non-created objects

Verifying `enterPin()`

Demo

`ATM.java::enterPin()`

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.

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

Further Reading

At `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*

Don't Be Always Formal . . .