Formal Methods for Software Development

Java Modeling Language, Part I

Wolfgang Ahrendt

28 September 2017

Role of JML in the Course

programming/modelling language	property/specification language	verification technique
PROMELA	LTL	model checking
JAVA	JML	deductive verification

Unit Specifications

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

instead:

unit specification – *contracts among implementers* on various levels:

- application level – application level
- application level – library level
- library level – library level

Unit Specifications

In the object-oriented setting:

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

We start with method specifications.

Method specifications *potentially* refer to:

- initial values of formal parameters
- result value
- prestate and poststatevisible part of pre/poststate

Specifications as Contracts

To stress the different roles – obligations – responsibilities in a specification:

widely used analogy of the specification as a *contract*

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

Contract between caller and callee (called method)

callee guarantees certain outcome *provided* caller guarantees prerequisites

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.*

Getting More Precise: Specification as Contract

Contract states what is guaranteed under which conditions.

precondition card is inserted, user not yet authenticated,
pin is correct
postcondition user is authenticated

precondition card is inserted, user not yet authenticated,
`wrongPINCounter < 2` and pin is incorrect
postcondition `wrongPINCounter` has been increased by `1`,
user is not authenticated

precondition card is inserted, user not yet authenticated,
`wrongPINCounter >= 2` and pin is incorrect
postcondition card is confiscated
user is not authenticated

Meaning of Pre/Postcondition 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.*

1. No guarantees are given when the precondition is not satisfied.
2. Termination may or may not be guaranteed.
3. In case of termination, it may be normal or abrupt.

non-termination and abrupt termination ⇒ next lecture

Formal Specification

Natural language specs are very important and widely used, we focus on

Formal Specification

Describe contracts 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
  - monitoring
  - 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.

⇒ JML is not external to JAVA

JML

is

JAVA + FO Logic + pre/postconditions, 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 by Example

from the file ATM.java

```
⋮

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

⋮
```

Everything between /* and */ is invisible for Java.

JML by Example

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

But:

A Java comment with '@' as its first character
it is *not* a comment for JML tools.

JML annotations appear in Java comments starting with @.

How about "//" comments?

JML by Example

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

equivalent to:

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

The easiest way to comment out JML:

```
/*␣@ public normal_behavior ... @*/
```

```
//␣@ public normal_behavior
//␣@ requires !customerAuthenticated;
...
```

JML by Example

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

What about the intermediate '@'s?

Within a JML annotation, a '@' is ignored:

- if it is the first (non-white) character in the line
- if it is the last character before '*/'.

⇒ The blue '@'s are not *required*, but it's a convention to use them.

JML by Example

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

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

1. it is accessible from all classes and interfaces
2. it can only mention `public` fields/methods of this class

2. Can be a problem. Solution later in the lecture.

JML by Example

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

Each keyword ending with **`behavior`** opens a ‘specification case’.

`normal_behavior` Specification Case

The method guarantees to *not* throw any exception (on the top level), *if the caller guarantees all preconditions of this specification case*.

JML by Example

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

This specification case has two preconditions (marked by **`requires`**)

1. `!customerAuthenticated`
2. `pin == insertedCard.correctPIN`

here:
preconditions are *boolean Java expressions*

in general:
preconditions are *boolean JML expressions* (see below)

JML by Example

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

specifies only the case where both preconditions are true in prestate

the above is equivalent to:

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

JML by Example

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

This specification case has one postcondition (marked by **ensures**)

- `customerAuthenticated`

here:
postcondition is *boolean Java expressions*

in general:
postconditions are *boolean JML expressions* (see below)

JML by Example

different specification cases are 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) {
    if ( ...
```

JML by Example

```
/*@ <spec-case1> also
  @
  @ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin != insertedCard.correctPIN;
  @ requires wrongPINCounter < 2;
  @ ensures wrongPINCounter == \old(wrongPINCounter) + 1;
  @*/
public void enterPIN (int pin) { ...
```

for the first time, JML expression not a JAVA expression

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

E can be any (arbitrarily complex) JML expression.

JML by Example

```
/*@ <spec-case1> also <spec-case2> also
  @
  @ public normal_behavior
  @ requires insertedCard != null;
  @ requires !customerAuthenticated;
  @ requires pin != insertedCard.correctPIN;
  @ requires wrongPINCounter >= 2;
  @ ensures insertedCard == null;
  @ ensures \old(insertedCard).invalid;
  @*/
public void enterPIN (int pin) { ...
```

two postconditions state that:

‘Given the above preconditions, `enterPIN` guarantees:

`insertedCard == null` and `\old(insertedCard).invalid`’

JML by Example

Question:

could it be

```
  @ ensures \old(insertedCard.invalid);
```

instead of

```
  @ ensures \old(insertedCard).invalid;
```

??

Specification Cases Complete?

consider spec-case-1:

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

what does spec-case-1 *not* tell about poststate?

recall: fields of class `ATM`:

```
insertedCard
customerAuthenticated
wrongPINCounter
```

what happens with `insertCard` and `wrongPINCounter`?

Completing Specification Cases

completing spec-case-1:

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

Completing Specification Cases

completing spec-case-2:

```
@ public normal_behavior
@ requires !customerAuthenticated;
@ requires pin != insertedCard.correctPIN;
@ requires wrongPINCounter < 2;
@ ensures wrongPINCounter == \old(wrongPINCounter) + 1;
@ ensures insertedCard == \old(insertedCard);
@ ensures customerAuthenticated
@         == \old(customerAuthenticated);
```

Completing Specification Cases

completing spec-case-3:

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

Assignable Clause

unsatisfactory to add

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

for all locations *loc* which *do not* change

instead:
add assignable clause for all locations which *may* change

```
@ assignable loc1,...,locn;
```

Meaning: No location other than $loc_1, \ldots, loc_n$ can be assigned to.

Special cases:

No location may be changed:

```
@ assignable \nothing;
```

Unrestricted, method allowed to change anything:

```
@ assignable \everything;
```

Specification Cases with Assignable

completing spec-case-1:

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

Specification Cases with Assignable

completing spec-case-2:

```
@ public normal_behavior
@ requires !customerAuthenticated;
@ requires pin != insertedCard.correctPIN;
@ requires wrongPINCounter < 2;
@ ensures wrongPINCounter == \old(wrongPINCounter) + 1;
@ assignable wrongPINCounter;
```

Specification Cases with Assignable

completing spec-case-3:

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

Assignable Groups

You can specify groups of locations as assignable, using '*'.

example:

```
@ assignable o.*, a[*];
```

makes all fields of object o and all positions of array a assignable.

Literature for this and the next Lecture

KeYbook *W. Ahrendt*, B. Beckert, R. Bubel, R. Hähnle, P. Schmitt, M. Ulbrich, editors.
Deductive Software Verification - The KeY Book
Vol 10001 of *LNCS*, Springer, 2016
(E-book at `link.springer.com`)

Essential reading:

New JML Tutorial M. Huisman, *W. Ahrendt*, D. Grahl, M. Hentschel.
Formal Specification with the Java Modeling Language
Chapter 7 in [KeYbook]

Further reading, all available at
`www.eecs.ucf.edu/~leavens/JML//index.shtml`