Testing, Debugging, and Verification

Formal Specification, Part II

Wolfgang Ahrendt, Vladimir Klebanov, Moa Johansson

21 November 2012

Recap: First Order Logic

- Signature Σ:
  - Types T_Σ
  - Functions (incl. constants) F_Σ
  - Predicates P_Σ
  - Typing function α
- Example Σ_1:
  - $T_1 = \{int, bool\}$
  - $F_1 = \{+, \ -\} \cup \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
  - $P_1 = \{<\}$
  - $\alpha(+) = \alpha(-) = (int, int) \rightarrow int$
    $\alpha(<) = (int, int) \rightarrow bool$
    $\ldots = \alpha(-1) = \alpha(0) = \alpha(1) \ldots$
  - In addition, set of (typed) variables V.

Recap: First Order Terms and Formulas

Terms are built from

- Functions
- Constants (functions with no arguments) and
- Variables
- E.g. $x + 2$, -5

Atomic formulas are boolean 'terms'.

- *true*, *false*
- Equalities: $t_1 = t_2$
- Predicates
- E.g. $x < y$, $x = 4$

Recap: Boolean Connectives

Formulas are built from (atomic) formulas combined with boolean connectives:

FOL	Meaning	Java (if applicable)
$\neg A$	not A	`!A`
$A \wedge B$	A and B	`A && B`
$A \vee B$	A or B	`A || B`
$A \rightarrow B$	A implies B	
$A \leftrightarrow B$	A is equivalent of B, A if and only if B	
$\forall\ \tau\ x.\ A$	For all x of type τ, A holds.	
$\exists\ \tau\ x.\ A$	There exists some x such that A holds.	

General Formulas: Examples

(signatures/types left out here)

Example (There exist at least two elements)

$\exists x, y; \neg(x = y)$

Example (Strict partial order)

Irreflexivity $\forall x; \neg(x < x)$

Asymmetry $\forall x; \forall y; (x < y \to \neg(y < x))$

Transitivity $\forall x; \forall y; \forall z;$
$(x < y \land y < z \to x < z)$

Example (All models have infinite domain)

Existence Successor $\forall x; \exists y; x < y$

In a real Logic Course ....

... we now would rigorously define:

- validity of formulas
- provability of formulas (in various calculi)

⇒ see course ‘Logic in Computer Science’

In *our* course, we stick to the intuitive meaning of formulas.

But we mention ‘models’ and ‘validity’.

Models, Validity, States

Model

A model assigns *meaning* to the symbols in $F_\Sigma \cup P_\Sigma$
(assigning functions to function symbols, relations to predicate symbols).

In a given model M, a closed formula is either true or not true.

Validity

A closed formula is valid if it is true in all models.

In the context of formal specification of imperative programs:
states[1] take over the role of models.

Which of the formulas in this slide set are true in some model?
Which of the formulas in this slide set are valid?

[1]together with input values and results, and possibly paired with old states

Useful Valid Formulas

Let ϕ and ψ be arbitrary, closed formulas (whether valid or not).

The following formulas are valid:

- $\neg(\phi \wedge \psi) \leftrightarrow \neg\phi \vee \neg\psi$
- $\neg(\phi \vee \psi) \leftrightarrow \neg\phi \wedge \neg\psi$
- $(\mathit{true} \wedge \phi) \leftrightarrow \phi$
- $(\mathit{false} \vee \phi) \leftrightarrow \phi$
- $\mathit{true} \vee \phi$
- $\neg(\mathit{false} \wedge \phi)$
- $(\phi \rightarrow \psi) \leftrightarrow (\neg\phi \vee \psi)$
- $\phi \rightarrow \mathit{true}$
- $\mathit{false} \rightarrow \phi$
- $(\mathit{true} \rightarrow \phi) \leftrightarrow \phi$
- $(\phi \rightarrow \mathit{false}) \leftrightarrow \neg\phi$

Useful Valid Formulas

Assume that x is the only variable which may appear freely in ϕ or ψ.

The following formulas are valid:

- $\neg(\exists\ \tau\ x;\ \phi) \leftrightarrow \forall\ \tau\ x;\ \neg\phi$
- $\neg(\forall\ \tau\ x;\ \phi) \leftrightarrow \exists\ \tau\ x;\ \neg\phi$
- $(\forall\ \tau\ x;\ \phi \land \psi) \leftrightarrow (\forall\ \tau\ x;\ \phi) \land (\forall\ \tau\ x;\ \psi)$
- $(\exists\ \tau\ x;\ \phi \lor \psi) \leftrightarrow (\exists\ \tau\ x;\ \phi) \lor (\exists\ \tau\ x;\ \psi)$

Are the following formulas also valid?

- $(\forall\ \tau\ x;\ \phi \lor \psi) \leftrightarrow (\forall\ \tau\ x;\ \phi) \lor (\forall\ \tau\ x;\ \psi)$
- $(\exists\ \tau\ x;\ \phi \land \psi) \leftrightarrow (\exists\ \tau\ x;\ \phi) \land (\exists\ \tau\ x;\ \psi)$

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

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 (but see last part of the course)
- ✔ ...
- ✘ ...

✔: 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

To not confuse JAVA compiler:

JML annotations live in in special comments,
ignored by JAVA, recognized by JML.

Recall: 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 () { ... }

}
```

Recall: 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 is authenticated.*

Recall: 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, `pin` is incorrect and `wrongPINCounter < 2`
postcondition	`wrongPINCounter` is increased by 1 user is not authenticated
precondition	card is inserted, user not yet authenticated, `pin` is incorrect and `wrongPINCounter >= 2`
postcondition	card is confiscated user is not authenticated

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 ( ....
⋮
```

JML by Example

```
/*@ 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.

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;
  @*/
public void enterPIN (int pin) {
    if ( ....
```

equivalent to:

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

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 on **`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 pre-state

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, a JML expression is not a JAVA expression

`\old(E)` means: *E* evaluated in the pre-state 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`'

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 post-state?

recall: fields of class `ATM`:

```
insertedCard
customerAuthenticated
wrongPINCounter
```

What happens with `insertedCard` 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 $loc_1, \dots, loc_n$ 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 wrongPINCounter,
@            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 locations of array a assignable.

JML Modifiers

JML extends the JAVA modifiers by additional modifiers.

The most important ones are:

- `spec_public`
- `pure`

Aim: admitting more class elements to be used in JML expressions.

JML Modifiers: spec_public

In enterPIN example, pre/post-conditions made heavy use of class fields

But: **public** specifications can only talk about **public** fields.

Not desired: make all fields public.

One solution:

- keep the fields **private/protected**
- make those needed for specification **spec_public**

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

JML Modifiers: `pure`

It can be handy to use method calls in JML annotations.
Examples:

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

allowed if, and only if method is guaranteed to have no side effects.

In JML, you can specify methods to be '**`pure`**':

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

'**`pure`**' puts obligation on implementer, not to cause side effects, but allows to use method in annotations

'**`pure`**' similar to '**`assignable \nothing;`**', but global to method

JML Expressions $\neq$ Java Expressions

`boolean` JML Expressions (to be completed)

- each side-effect free **`boolean`** JAVA expression is a **`boolean`** JML expression
- if `a` and `b` are **`boolean`** JML expressions, and `x` is a variable of type `t`, then the following are also **`boolean`** JML expressions:
  - `!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”)
  - ...
  - ...
  - ...
  - ...

Beyond `boolean` Java expressions

How to express the following?

- An array `arr` only holds values ≤ 2
- The variable `m` holds the maximum entry of array `arr`
- All `Account` objects in the array `bank` are stored at the index corresponding to their respective `accountNumber` field
- All created instances of class `BankCard` have different `cardNumbers`

First-order Logic in JML Expressions

JML **`boolean`** expressions extend JAVA **`boolean`** expressions by:

- implication
- equivalence
- quantification

boolean JML Expressions

boolean JML expressions are defined recursively:

boolean JML Expressions

- each side-effect free **boolean** JAVA expression is a **boolean** JML expression
- if `a` and `b` are **boolean** JML expressions, and `x` is a variable of type `t`, then the following are also **boolean** JML expressions:
  - `!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")

JML Quantifiers

In

`(\forall t x; a; b)`

`(\exists t x; a; b)`

a called "range predicate"

those forms are redundant:

`(\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

`(\forall t x; a; b)` and `(\exists t x; a; b)`

widely used

pragmatics of range predicate:

`a` used to restrict range of `x` further than `t`

Example: "`arr` is sorted at indexes between 0 and 9":

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