Testing, Debugging, and Verification

Formal Specification, Part I

Wolfgang Ahrendt, Vladimir Klebanov

19 November 2012

Where are we in the course?

— past course parts —

✔ Testing

✔ Debugging

— upcoming course parts —

- Formal Specification (starting today)
- Test Case Generation
- Formal Program Verification

This Part

Theme: Formal Specification

Structure:

- three lectures (19, 21 (x2)/11)
- one exercise (Monday 26/11)
- one lab assignment

Content:

- Specification as such
- Unit Specification
- Formal Unit Specification

Motivation

As motivating examples, let's consider two programs.

Example 1: method `alwaysTrue()`

```
// should always return true
public static boolean alwaysTrue(int i) {

    // Just 'return true;' is all too boring.
    // Instead:
    return ( Math.abs(i) >= 0 );

}
```

Example 1: Testing `alwaysTrue()`

```
Scanner sc = new Scanner(System.in);

while (true) {

    // read an integer from System.in
    int i = sc.nextInt();

    // this will print "true"
    System.out.println(alwaysTrue(i));
}
```

Demo: `TestAlwaysTrue.java`

Surprise: with input -2147483648, the program prints `false`!

We want to understand the problem

- Another test:
  `System.out.println(Math.abs(-2147483648))`
  prints
  `-2147483648`
- We cannot come any closer to the problem by testing/debugging.
- So how can we?

Specification is the Answer!

From the Java API Specification, class Math:

`public static int abs(int a)`

Returns the absolute value of an int value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.

Note that if the argument is equal to the value of `Integer.MIN_VALUE`, the most negative representable `int` value, the result is that same value, which is negative.

Caller and Callee disagree

The problem was:

Caller (here `alwaysTrue()`)

had unfulfilled expectations about

Callee (here `Math.abs()`).

Example 2: `equal` Objects in `Sets`

```
public class Book {

    private String title;
    private String author;
    private long isbn;

    public Book(...) { ... }
    }

    public boolean equals(Object obj) {
        if (obj instanceof Book) {
            Book other = (Book) obj;
            return (isbn == other.isbn);
        }
        return false;
    }

    public String toString() { ... }
}
```

Example 2: `equal` Objects in `Sets`

From the Java API Specification, Interface Set:

`public interface Set`
`extends Collection`

Sets contain no pair of elements e1, e2 such that `e1.equals(e2)` ...

...

`boolean add(E e)`

Adds `e` to this set if the set contains no element `e2` such that `e.equals(e2)` ...

Example 2: `equal` Objects in `Sets`

Adding two equal books to a set:

```
Set catalogue = new HashSet();

Book b1 = new Book("Effective␣Java",
                   "Joshua␣Bloch",
                   201310058);

Book b2 = new Book("Effective␣Java",
                   "J.␣Bloch",
                   201310058);

catalogue.add(b1);
catalogue.add(b2);
```

How many elements has `catalogue` now?

Demo: `AddTwoBooks.java`

two!(?)

We want to understand the problem also

But again:

- We cannot come any closer to the problem by testing/debugging.
- So how can we?

Again: Specification is the Answer!

- Here, specification of `Set` or `HashSet` does not reveal problem
- Instead: check the specification of `Book`!
- Is there any?
- Yes, because `Book` extends `Object`, and inherits the specifications from there!

Checking the API of `Object`

`public int hashCode()`

...
If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
...

By overriding `equals` only, and not `hashCode`, we broke the specification of `Book.hashCode()`.

Caller and Callee disagree

The problem was:

Caller (here `HashSet.add()`)
had unfulfilled expectations about
Callee (here `Book.hashCode()`).

Here:
The caller is library code, the callee is a method from our own class!

⇒ Call Back Mechanism in OO Programming

Is this nasty?

- How could the implementer of `Book` foresee whether some class implementing `Set` would call `Book.hashCode()`?
- He/she cannot!

No alternative to fulfilling the inherited specification of `Object`, as potential callers might rely on it in unforeseeable ways!

Demo: `fixing AddTwoBooks.java`

Example1/2: Similarities and Differences

In both cases:
caller had unfulfilled expectations about callee

Difference: who is to blame?

Example 1: the caller (`alwaysTrue()`)

Example 2: the callee (`Book.hashCode()`)

We will focus on a crystal clear distinction

- of these different roles, and
- the different obligations attached to either of the roles.

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

Contract between *caller* and *callee* of method

callee guarantees certain outcome *provided* caller guarantees prerequisites

What kind of 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 $\leftrightarrow$ application level
- application level $\leftrightarrow$ library level
- library level $\leftrightarrow$ library level

What kind of Specifications

Natural language specs are very important (see the examples above).

Still:
we focus on

"Formal" specifications:

Describing contracts of units in a mathematically precise language.

Motivation:

- higher degree of precision.
- eventually, automation of program analysis of various kinds:
  - static checking
  - test case generation
  - program verification

Unit Specifications

In the object-oriented setting:

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

First, focus on methods

Methods can be specified by referring to for example:

- result value,
- initial values of formal parameters,
- pre-state and post-state (accessible part)

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, `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

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.*

1. No guarantees are given 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 (cf. exceptions).

In the above: what do we mean by *state*?

Prerequisite: Object-oriented States

By state, we mean a 'snapshot' of the system, at any point during the the computation.

An *object oriented* state consists of:

- the set $\mathcal{C}$ of all loaded classes
- the values of the static fields of classes in $\mathcal{C}$
- the set $\mathcal{O}$ of references to all created objects
- the values of the instance fields of objects in $\mathcal{O}$

Here, values of *local variables* and *formal parameters* are *not* considered part of the state.

Prerequisite: Accessible State

Like implementations, specifications can only refer to the locally accessible part of the state (e.g., not to `private` fields of other classes).

Prerequisite: Accessible State

In our context, we stick to the following principle:

Same Accessible State for Specifications and Implementations:

In a given local context, specifications and implementations can access the same part of the overall state.[a]

[a] Later, we'll refine this principle, and introduce well defined exceptions.

Specifications talk only about those parts of the state which are accessible by:

- respecting JAVA's accessibility rules (`public`, `protected`, `private`),
- following (accessible) references, starting from local fields.

Stateless Specification

A stateless specification of a (non-`void`) method talks

- only about
  - the result of a call
  - the initial value of input parameters
- but not about
  - (any part of) the state

examples:

interface/class:	*method:*
`Math`	`static int abs(int a)`
`Math`	`static double sqrt(double a)`

Stateless Specification: `Math.abs()`

from the JAVA API:

Specification of `static int abs(int a)`

Returns the absolute value of an int value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.
Note that if the argument is equal to the value of `Integer.MIN_VALUE`, the most negative representable int value, the result is that same value, which is negative.

Green: Intuitive description rather than a specification.
Red: Precise specification by case distinction, given we know what 'negative' and 'negation' mean exactly.
Blue: A consequence of the specification, i.e., a *redundant part* of it.

Red and Blue are candidates for formalisation.

Going a bit more formal

```
static int abs(int a)
```

Informal spec:
If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.

Semi formal:

- Under the precondition '$a \in [0 \ldots 2147483647]$', **abs** ensures the postcondition 'result $=$ a'.
- Under the precondition '$a \in [-2147483648 \ldots -1]$', **abs** ensures the postcondition 'result $= -a$'.

Going a bit more formal

`static int` **`abs`**`(int a)`

Redundant informal spec:
Note that if the argument is equal to the value of `Integer.MIN_VALUE`, the most negative representable int value, the result is that same value, which is negative.

Semi formal:

- Under the precondition 'a $= -2147483648$',
  `abs` ensures the postcondition 'result $= -2147483648$'.

Or simply:[1]

- **`abs`**$(-2147483648) = -2147483648$

[1] But be careful when using a method call in a specification, see below.

State Aware Specification

A state aware specification of a (`void` or non-`void`) method talks about

- result of a call (if non-`void`)
- initial value of input parameters
- *two* states:
  - 'pre-state' of method call
  - 'post-state' of method call

Examples:

interface/class:	*method:*
`List`	`Object set(int index, Object element)`

State Aware Specification: `List.set(i,e)`

from the Java API of `List.set` *(simplified):*

`public Object` **`set`**`(int index, Object element)`

Replaces the element at the specified position in this list with the specified element.

Parameters:
`index` - index of element to replace.
`element` - element to be stored at the specified position.
Returns:
the element previously at the specified position.
Throws:
`IndexOutOfBoundsException`
- if the index is out of range (index $<$ 0 || index $>=$ size()).

Why is the spec state aware?
Talks about state, state change

Going a bit more formal

`public Object` **`set`**`(int index, Object element)`

Informal spec:
Replaces the element at the specified position in this list with the specified element.

Semi formal:
`set` ensures the following postcondition:

- `element` = '`get(index)` *evaluated in post-state*'

Does this capture the meaning of the word 'replace'?

Going a bit more formal

`public Object` **`set`**`(int index, Object element)`

Informal spec:
Replaces the element at the specified position in this list with the specified element.

Semi formal:
`set` ensures the following postconditions:

- `element` = '`get(index)` *evaluated in post-state*'
- for all `j` $\in [0 \ldots$`size()` $- 1]$ with `j` $\neq$ `index`:
  '`get(j)` *in post-state*' = '`get(j)` *in pre-state*'

Going a bit more formal

`public Object set(int index, Object element)`

Informal spec:
Replaces the element at the specified position in this list with the specified element ... Returns the element previously at the specified position.

Semi formal:
`set` ensures the following postconditions:

- result = '`get(index)` *evaluated in pre-state*'
- `element` = '`get(index)` *evaluated in post-state*'
- for all `j` $\in [0...$`size()` $- 1]$ with `j` $\neq$ `index`:
  '`get(j)` *in post-state*' = '`get(j)` *in pre-state*'

Going a bit more formal

`public Object set(int index, Object element)`

Informal spec:
Replaces the element at the specified position in this list with the specified element ... Returns the element previously at the specified position ... Throws `IndexOutOfBoundsException` if the index is out of range (index $<$ 0 || index $>=$ size()).

Semi formal:

- Under the precondition '`index` $\in$ [0...`size()` $-$ 1]', **`set`** ensures the following postconditions:
  - result = '`get(index)` *evaluated in pre-state*', and
  - `element` = '`get(index)` *evaluated in post-state*', and
  - for all `j` $\in$ [0...`size()` $-$ 1] with `j` $\neq$ `index`: '`get(j)` *in post-state*' = '`get(j)` *in pre-state*'
- Under the precondition '`index` $\notin$ [0...`size()` $-$ 1]', **`set`** throws `IndexOutOfBoundsException`.

Reflection

Desired features of framework for *formal specification*:

- Pairs of
  - preconditions
  - postconditions
- A language to express these conditions, capturing:
  - relations, equality, logical connectives
  - *quantification*
- Constructs to refer to:
  - values in *new* and in *old* state
  - throwing of exceptions

A Formal Language for Specifications

We will use the 'Java Modelling Language' (JML) to specify JAVA programs.

JML combines

- JAVA
- First-Order Logic

In the following, we introduce:

1. First-Order Logic
2. JML

First-Order Logic: Signature

Signature

A first-order signature Σ consists of

- a set T_Σ of types
- a set F_Σ of function symbols
- a set P_Σ of predicate symbols
- a typing α_Σ

Intuitively, the typing-function α_Σ determines

- for each function and predicate symbol:
  - its arity, i.e., number of arguments
  - its argument types
- for each function symbol its result type.

formally:

- $\alpha_\Sigma(p) \in {T_\Sigma}^*$ for all $p \in P_\Sigma$ (arity of p is $|\alpha_\Sigma(p)|$)
- $\alpha_\Sigma(f) \in {T_\Sigma}^* \times T_\Sigma$ for all $f \in F_\Sigma$ (arity of f is $|\alpha_\Sigma(f)| - 1$)

Example Signature 1 + Constants

$T_{\Sigma_1} = \{\texttt{int}\}$,
$F_{\Sigma_1} = \{\texttt{+}, \texttt{-}\} \cup \{..., \texttt{-2}, \texttt{-1}, \texttt{0}, \texttt{1}, \texttt{2}, ...\}$,
$P_{\Sigma_1} = \{\texttt{<}\}$

$\alpha_{\Sigma_1}(\texttt{<}) = (\texttt{int},\texttt{int} \rightarrow \texttt{boolean})$
$\alpha_{\Sigma_1}(\texttt{+}) = \alpha_{\Sigma_1}(\texttt{-}) = (\texttt{int},\texttt{int} \rightarrow \texttt{int})$
$\alpha_{\Sigma_1}(\texttt{0}) = \alpha_{\Sigma_1}(\texttt{1}) = \alpha_{\Sigma_1}(\texttt{-1}) = ... = (\texttt{int})$

Constants

A function symbol `f` with $|\alpha_{\Sigma_1}(f)| = 1$ (i.e., with arity 0) is called *constant symbol*.

here, the constants are: $..., \texttt{-2}, \texttt{-1}, \texttt{0}, \texttt{1}, \texttt{2}, ...$

Example Signature 1 + Notation

$T_{\Sigma_1} = \{\texttt{int}\},$
$F_{\Sigma_1} = \{\texttt{+}, \texttt{-}\} \cup \{..., \texttt{-2}, \texttt{-1}, \texttt{0}, \texttt{1}, \texttt{2}, ...\},$
$P_{\Sigma_1} = \{\texttt{<}\}$

$\alpha_{\Sigma_1}(\texttt{<}) = (\texttt{int},\texttt{int}\rightarrow\texttt{boolean}\ )$
$\alpha_{\Sigma_1}(\texttt{+}) = \alpha_{\Sigma_1}(\texttt{-}) = (\texttt{int},\texttt{int}\rightarrow\texttt{int})$
$\alpha_{\Sigma_1}(\texttt{0}) = \alpha_{\Sigma_1}(\texttt{1}) = \alpha_{\Sigma_1}(\texttt{-1}) = ... = (\texttt{int})$

Notation

Notation for typing of *functions*:
write $(\tau_1, ..., \tau_n \rightarrow \tau_{n+1})$, where return type is after arrow.

Example Signature 2

$T_{\Sigma_2} = \{\texttt{int},\ \texttt{LinkedIntList}\},$
$F_{\Sigma_2} = \{\texttt{null},\ \texttt{new},\ \texttt{elem},\ \texttt{next}\} \cup \{\texttt{...,-2,-1,0,1,2,...}\}$
$P_{\Sigma_2} = \{\}$

intuitively, `elem` and `next` model fields of `LinkedIntList` objects

$\alpha_{\Sigma_2}(\texttt{null}) = (\texttt{LinkedIntList})$
$\alpha_{\Sigma_2}(\texttt{new}) = (\texttt{int}, \texttt{LinkedIntList} \rightarrow \texttt{LinkedIntList})$
$\alpha_{\Sigma_2}(\texttt{elem}) = (\texttt{LinkedIntList} \rightarrow \texttt{int})$
$\alpha_{\Sigma_2}(\texttt{next}) = (\texttt{LinkedIntList} \rightarrow \texttt{LinkedIntList})$

and as before:
$\alpha_{\Sigma_2}(0) = \alpha_{\Sigma_2}(1) = \alpha_{\Sigma_2}(\texttt{-1}) = ... = (\texttt{int})$

First-Order Terms

We assume a set V of variables ($V \cap (F_\Sigma \cup P_\Sigma) = \emptyset$).
Each $v \in V$ has a unique type $\alpha_\Sigma(v) \in T_\Sigma$.

Terms are defined recursively:

Terms

A first-order term of type $\tau \in T_\Sigma$

- is either a variable of type τ, or
- has the form $f(t_1, \ldots, t_n)$,
  where $f \in F_\Sigma$ has result type τ, and each t_i is term of the correct type, following the typing α_Σ of f.

If f is a constant symbol, we write f instead of $f()$.

Terms over Signature 1

Example terms over Σ_1:
(assume variables v_1, v_2 with $\alpha_{\Sigma_1}(v_1) = \alpha_{\Sigma_1}(v_2) = \texttt{int}$)

- -7
- $+(-2,\ 99)$
- $-(7,\ 8)$
- $+(-(7,\ 8),\ 1)$
- $+(-(v_1,\ 8),\ v_2)$

Some variants of FOL allow infix notation of functions:

- $-2\ +\ 99$
- $7\ -\ 8$
- $(7\ -\ 8)\ +\ 1$
- $(v_1\ -\ 8)\ +\ v_2$

Terms over Signature 2

Example terms over Σ_2:
(assume variables o with $\alpha_{\Sigma_2}(o) =$ `LinkedIntList`,
assume variable v with $\alpha_{\Sigma_2}(v) =$ `int`)

- `-7`
- `null`
- `new(13, null)`
- `elem(new(13, null))`
- `next(next(o))`

for first-order functions modeling object fields,
we allow dotted postfix notation:

- `new(13, null).elem`
- `o.next.next`

Atomic Formulas

Logical Atoms

Given a signature Σ.
A logical atom has either of the forms

- *true*
- *false*
- $t_1 = t_2$ *("equality")*,
  where t_1 and t_2 have the same type.
- $p(t_1, \ldots, t_n)$ *("predicate")*,
  where $p \in P_\Sigma$, and each t_i is term of the correct type,
  following the typing α_Σ of p.

Atomic Formulas over Signature 1

example formulas over Σ_1:
(assume variables v with $\alpha_{\Sigma_1}(v) = \texttt{int}$)

- $7 = 8$
- $7 < 8$
- $-2 - v < 99$
- $v < (v + 1)$

Atomic Formulas over Signature 2

example formulas over Σ_2:
(assume variables o with $\alpha_{\Sigma_2}(o) = \texttt{LinkedIntList}$,
assume variable v with $\alpha_{\Sigma_2}(v) = \texttt{int}$)

- $\texttt{new}(13, \texttt{null}) = \texttt{null}$
- $\texttt{elem}(\texttt{new}(13, \texttt{null})) = 13$
- $\texttt{next}(\texttt{new}(13, \texttt{null})) = \texttt{null}$
- $\texttt{next}(\texttt{next}(o)) = o$

General Formulas

first-order formulas are defined recursively:

Formulas

- each atomic formula is a formula
- with ϕ and ψ formulas, x a variable, and τ a type, the following are also formulas:
  - $\neg\phi$ *("not ϕ")*
  - $\phi \wedge \psi$ *("ϕ and ψ")*
  - $\phi \vee \psi$ *("ϕ or ψ")*
  - $\phi \rightarrow \psi$ *("ϕ implies ψ")*
  - $\phi \leftrightarrow \psi$ *("ϕ is equivalent to ψ")*
  - $\forall\ \tau\ x;\ \phi$ *("for all x of type τ holds ϕ")*
  - $\exists\ \tau\ x;\ \phi$ *("there exists an x of type τ such that ϕ")*

In $\forall\ t\ x;\ \phi$ and $\exists\ t\ x;\ \phi$ the variable x is 'bound' (i.e., 'not free'). Formulas with no free variable are 'closed'.