# Testing, Debugging, and Verification

## Formal Specification, Part I

Srinivas Pinisetty[1]

19 November 2018

[1] Lecture slides based on material from Wolfgang Aherndt,..

# Where are we in the course?

— past course parts —

- ✔ Testing
- ✔ Debugging

— upcoming course parts —

- Formal Specification (starting today)
- Formal Program Verification (theory behind)
- Loop Invariant Generation

# This Part

Formal Specification

**Structure**

- three lectures
- one exercise
- one hand-in lab assignment

# Formal Specification: Contents

## Content

- Why specification is important.
- Writing formal specifications: First Order Logic.
- **Dafny:** A programming language with support for automated checking of formal specifications.
- Dafny supports automated checking of method pre- and postconditions.
- **Note:** Focus is on writing good specifications, not so much programming.
- With skills in Java, simple programming in Dafny is very similar.

# 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

# 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)
```

# 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.

# 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)` ...

...

# 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<Book> catalogue = new HashSet<Book>();

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`

# Example 2: equal Objects in Sets

Adding two equal books to a set:

```
Set<Book> catalogue = new HashSet<Book>();

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..

- Here, specification of `Set` or `HashSet` does not reveal problem
- Instead: check the specification of `Book`!
- Is there any?

# We want to understand the problem..

- 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.

...

# 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!

# Example1/2: Similarities and Differences

In both cases:
caller had unfulfilled expectations about callee

Difference: who is to blame?

# 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()`)

# 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

# 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

# Formal 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.

# Formal 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.
- Automation of program analysis of various kinds:
  - formal verification
  - test case generation

# A first glance at Dafny

- Object oriented, like Java.
- Designed to make it easy to write correct code.
- Write specification in formal language (annotations specifying program behaviour).
- Automatically proves that the code matches annotations.
- Also proves absence of run time errors, e.g. null dereferencing, index-out-of-bounds etc.
- We will look at Dafny in more detail in the coming lectures.

Knowledge about formal specification/verification is useful (enables precise thinking), even if you will not regularly use Isabelle/Dafny/Coq/etc.

# Example: ATM.dfy

```
class ATM {

    // fields:
    var insertedCard : BankCard;
    var wrongPINCounter : int;
    var customerAuthenticated : bool;

    // methods:
    method insertCard (card : BankCard) { ... }
    method enterPIN (pin :  int) { ... }
    ...

}
```

# Informal Specification

Very informal specification of '`enterPIN (pin:int)`':

> *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 invalidated and 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.

`enterPIN (pin:int)`

# Getting More Precise: Specification as Contract

Contract states what is guaranteed under which conditions.

`enterPIN (pin:int)`

*precondition* card is inserted, user not yet authenticated,

# Getting More Precise: Specification as Contract

Contract states what is guaranteed under which conditions.

`enterPIN (pin:int)`

*precondition* card is inserted, user not yet authenticated,

*postcondition* If `pin` is correct, then the user is authenticated

# Getting More Precise: Specification as Contract

Contract states what is guaranteed under which conditions.

`enterPIN (pin:int)`

| | |
|---|---|
| *precondition* | card is inserted, user not yet authenticated, |
| *postcondition* | If `pin` is correct, then the user is authenticated |
| *postcondition* | If `pin` is incorrect and `wrongPINCounter < 2` then `wrongPINCounter` is increased by 1 and user is not authenticated |

# Getting More Precise: Specification as Contract

Contract states what is guaranteed under which conditions.

`enterPIN (pin:int)`

| | |
|---|---|
| *precondition* | card is inserted, user not yet authenticated, |
| *postcondition* | If `pin` is correct, then the user is authenticated |
| *postcondition* | If `pin` is incorrect and `wrongPINCounter < 2` then `wrongPINCounter` is increased by 1 and user is not authenticated |
| *postcondition* | If `pin` is incorrect and `wrongPINCounter >= 2` then card is confiscated and user is not authenticated |

# Getting More Precise: Specification as Contract

Contract states what is guaranteed under which conditions.

`enterPIN (pin:int)`

| | |
|---|---|
| *precondition* | card is inserted, user not yet authenticated, |
| *postcondition* | If `pin` is correct, then the user is authenticated |
| *postcondition* | If `pin` is incorrect and `wrongPINCounter < 2` then `wrongPINCounter` is increased by 1 and user is not authenticated |
| *postcondition* | If `pin` is incorrect and `wrongPINCounter >= 2` then card is confiscated and user is not authenticated |

Implicit preconditions in natural language spec: inserted card is not null, the card is valid. Should also be formalised!

# Mini Quiz: Identifying pre- and postconditions

The method **`insertCard(card:BankCard)`** has the following informal specification:

> *Inserts a bank card into the ATM if the card slot is free and provided the card is valid.*

- Identify at least two preconditions and at least one postcondition.
- Optional: think of sensible additional ones, not mentioned explicitly by the informal specification.

```
class ATM {
    var insertedCard : BankCard;
    var wrongPINCounter : int;
    var customerAuthenticated : bool; . . .}
```

# Mini Quiz: Identifying pre- and postconditions

The method **`insertCard(card:BankCard)`** has the following informal specification:

> *Inserts a bank card into the ATM if the card slot is free and provided the card is valid.*

- Identify at least two preconditions and at least one postcondition.
- Optional: think of sensible additional ones, not mentioned explicitly by the informal specification.

```
class ATM {
    var insertedCard : BankCard;
    var wrongPINCounter : int;
    var customerAuthenticated : bool; . . .}
```

preconditions:
ATM card slot is free.
The card is valid.
(The card is not null)

postconditions:
The ATM card slot is occupied.
(The user is not authenticated.)

# Reflection

How do we express pre- and postconditions formally?

Need a formal language to express:

- Set of
  - preconditions
  - postconditions
- A language to express these conditions, capturing:
  - relations, equality, logical connectives
  - *quantification*

Before diving in to Dafny:
Pause and learn a bit about First Order Logic.

# Recall: Propositional Logic

A propositional logic formula is built from

- Constants: true, false
- Boolean variables: P, Q, R... (atomic propositions)
- Connectives: $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$

| **Connective** | **Meaning** | **Dafny** |
|---|---|---|
| $\neg P$ | not P | `!P` |
| $P \wedge Q$ | P and Q | `P && Q` |
| $P \vee Q$ | P or Q | `P \|\| Q` |
| $P \rightarrow Q$ | P implies Q | `P ==> Q` |
| $P \leftrightarrow Q$ | P is equivalent to Q | `P <==> Q` |

Example: *"If you are a bunny, then you eat carrots."*

P: You are a bunny.

Q: You eat carrots.

# Recall: Propositional Logic

A propositional logic formula is built from

- Constants: true, false
- Boolean variables: P, Q, R... (atomic propositions)
- Connectives: $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$

| **Connective** | **Meaning** | **Dafny** |
|---|---|---|
| $\neg P$ | not P | `!P` |
| $P \wedge Q$ | P and Q | `P && Q` |
| $P \vee Q$ | P or Q | `P \|\| Q` |
| $P \rightarrow Q$ | P implies Q | `P ==> Q` |
| $P \leftrightarrow Q$ | P is equivalent to Q | `P <==> Q` |

Example: *"If you are a bunny, then you eat carrots."*

P: You are a bunny.

Q: You eat carrots.

$P \rightarrow Q$ : *"If you are a bunny, then you eat carrots."*

# Recall: Propositional Logic

Truth table:

| P | Q | P $\rightarrow$ Q |
|---|---|---|
| T | T | T |
| T | F | **F** |
| F | T | T |
| F | F | T |

# Recall: Propositional Logic

Truth table:

| P | Q | P $\rightarrow$ Q |
|---|---|---|
| T | T | T |
| T | F | **F** |
| F | T | T |
| F | F | T |

A formula $F$ is:

- Satisfiable if $F$ can be true.
- Valid if $F$ is always true.

# Recall: Propositional Logic

Truth table:

| P | Q | P $\rightarrow$ Q |
|---|---|---|
| T | T | T |
| T | F | **F** |
| F | T | T |
| F | F | T |

A formula $F$ is:

- Satisfiable if $F$ can be true.
- Valid if $F$ is always true.

**Exercise**:

Draw the truth-table for $\neg P \vee Q$. Do you notice anything interesting?

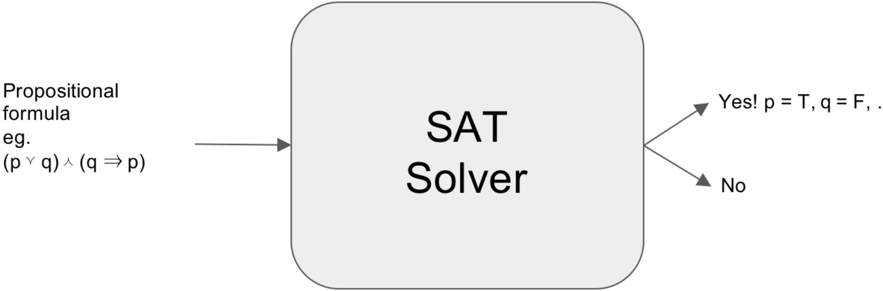
# Recall: Propositional Logic

## Some tautologies

- $\neg\neg\varphi \leftrightarrow \varphi$
- $\neg(\varphi \wedge \psi) \leftrightarrow \neg\varphi \vee \neg\psi$
- $\neg(\varphi \vee \psi) \leftrightarrow \neg\varphi \wedge \neg\psi$
- $\ldots$

# Propositional Satisfiability Problem (SAT Solver)

Given propositional logic formula, check whether it is satisfiable, and return a solution if it is.




- Program that solves whether a formula $F$ satisfiable.
- can be also used to check for validity of $F$ (if $\neg F$ is not satisfiable).
- Try during exercise session !!

# First-Order Logic (FOL)

Extends propositional logic by:

- **Types**, other than boolean
  e.g. int, real, BankCard, ....
- **Functions** (mathematical)
  e.g. $+$, max, abs, fibonacci,...
  - **Constants** are functions with no arguments
    e.g. 0, 1, fluffy
- **Predicates** (functions returning a boolean)
  e.g. isEven, $>$, isPrime...
- **Quantifiers**
  for all ($\forall$), there exists ($\exists$)

# First Order Logic: Syntax

## Terms

$$t ::= x | c | f(t_1, \cdots, t_n)$$

$x$ is any variable symbol, $c$ is any constant, $f$ is any function symbol of some arity $n$.

## Formulas

$$\begin{aligned} \phi ::= & \ P(t_1, \cdots, t_n) \\ & |(\phi \wedge \phi)|(\phi \vee \phi)|(\neg \phi)| \cdots \\ & |(\forall x : \phi)|(\exists x : \phi) \end{aligned}$$

$P$ is any predicate symbol of some arity $n$ and $t_i$ are terms.

# First Order Logic: Terms and Formulas

Terms are built from

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

# First Order Logic: Terms and Formulas

Terms are built from

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

Atomic formulas

- *true*, *false*
- Predicates
- E.g. $x < y$, $isPrime(2)$, $t_1 = t_2$

# First Order Logic: Terms and Formulas

Terms are built from

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

Atomic formulas

- *true*, *false*
- Predicates
- E.g. $x < y$, $isPrime(2)$, $t_1 = t_2$

FO Formulas are built recursively from atomic formulas and boolean connectives. E.g.

- $(x < y \wedge x = 4) \rightarrow 0 < (y - 4)$
- $\forall i : int.\ isEven(i) \rightarrow isOdd(i + 1)$

# Quantifiers

| Connective | Meaning/Dafny |
|---|---|
| $\forall x : t.\ P$ | *For all x of type t, P holds* |
| | In Dafny: `forall x :  t ::  P` |

# Quantifiers

| Connective | Meaning/Dafny |
|---|---|
| $\forall x : t.\ P$ | *For all x of type t, P holds* |
| | In Dafny: `forall x : t :: P` |
| $\exists x : t.\ P$ | *There exist an x of type t such that P holds* |
| | In Dafny: `exists x : t :: P` |

# Quantifiers

| Connective | Meaning/Dafny |
| --- | --- |
| $\forall x : t.\ P$ | *For all x of type t, P holds* |
| | In Dafny: `forall x : t :: P` |
| $\exists x : t.\ P$ | *There exist an x of type t such that P holds* |
| | In Dafny: `exists x : t :: P` |

Example: All entries in the array a are greater than 0

$\forall i : int.\ 0 \leq i < a.Length \rightarrow a[i] > 0$

# Quantifiers

| Connective | Meaning/Dafny |
|---|---|
| $\forall x : t.\ P$ | *For all x of type t, P holds* |
| | In Dafny: `forall x :  t ::  P` |
| $\exists x : t.\ P$ | *There exist an x of type t such that P holds* |
| | In Dafny: `exists x :  t ::  P` |

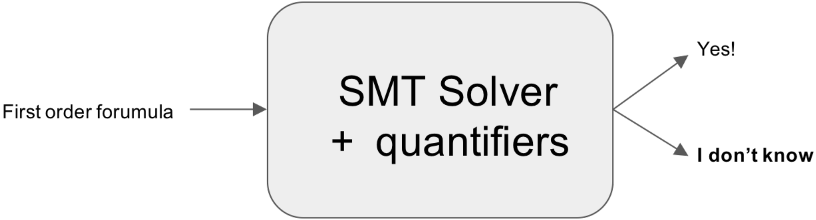
Example: All entries in the array a are greater than 0

$\forall i : int.\ 0 \leq i < a.Length \rightarrow a[i] > 0$

Example: There is at least one prime number in the array a

$\exists i : int.\ 0 \leq i < a.Length \wedge isPrime(a[i])$

# Satisfiablity modulo theories + Quantifiers




- Semi-decidable problem (often gives good results).

# Validity

A first order logic formula is valid if it is true in every interpretation (however we interpret the functions and constants)

# Valid Formulas

The following formulas are valid:

1. $\neg(\exists\, x : t.\ \phi) \leftrightarrow \forall\, x : t.\ \neg\phi$
2. $\neg(\forall\, x : t.\ \phi) \leftrightarrow \exists\, x : t.\ \neg\phi$
3. $(\forall\, x : t.\ \phi \wedge \psi) \leftrightarrow (\forall\, x : t.\ \phi) \wedge (\forall\, x : t.\ \psi)$
4. $(\exists\, x : t.\ \phi \vee \psi) \leftrightarrow (\exists\, x : t\ \phi) \vee (\exists\, x : t.\ \psi)$

# Valid Formulas

The following formulas are valid:

1. $\neg(\exists\, x : t.\ \phi) \leftrightarrow \forall\, x : t.\ \neg\phi$
2. $\neg(\forall\, x : t.\ \phi) \leftrightarrow \exists\, x : t.\ \neg\phi$
3. $(\forall\, x : t.\ \phi \wedge \psi) \leftrightarrow (\forall\, x : t.\ \phi) \wedge (\forall\, x : t.\ \psi)$
4. $(\exists\, x : t.\ \phi \vee \psi) \leftrightarrow (\exists\, x : t\ \phi) \vee (\exists\, x : t.\ \psi)$

Are the following formulas also valid?

- $(\forall\, x : t.\ \phi \vee \psi) \leftrightarrow (\forall\, x : t.\ \phi) \vee (\forall\, x : t.\ \psi)$
- $(\exists\, x : t.\ \phi \wedge \psi) \leftrightarrow (\exists\, x : t.\ \phi) \wedge (\exists\, x : t.\ \psi)$

# Valid Formulas

The following formulas are valid:

1. $\neg(\exists\ x : t.\ \phi) \leftrightarrow \forall\ x : t.\ \neg\phi$
2. $\neg(\forall\ x : t.\ \phi) \leftrightarrow \exists\ x : t.\ \neg\phi$
3. $(\forall\ x : t.\ \phi \wedge \psi) \leftrightarrow (\forall\ x : t.\ \phi) \wedge (\forall\ x : t.\ \psi)$
4. $(\exists\ x : t.\ \phi \vee \psi) \leftrightarrow (\exists\ x : t\ \phi) \vee (\exists\ x : t.\ \psi)$

Are the following formulas also valid?

- $(\forall\ x : t.\ \phi \vee \psi) \leftrightarrow (\forall\ x : t.\ \phi) \vee (\forall\ x : t.\ \psi)$
  - No! On the left, each x must make either $\phi$ or $\psi$ true. On the right, one of $\phi$ or $\psi$ must hold for every x.
- $(\exists\ x : t.\ \phi \wedge \psi) \leftrightarrow (\exists\ x : t.\ \phi) \wedge (\exists\ x : t.\ \psi)$

# Valid Formulas

The following formulas are valid:

1. $\neg(\exists\, x : t.\ \phi) \leftrightarrow \forall\, x : t.\ \neg\phi$
2. $\neg(\forall\, x : t.\ \phi) \leftrightarrow \exists\, x : t.\ \neg\phi$
3. $(\forall\, x : t.\ \phi \wedge \psi) \leftrightarrow (\forall\, x : t.\ \phi) \wedge (\forall\, x : t.\ \psi)$
4. $(\exists\, x : t.\ \phi \vee \psi) \leftrightarrow (\exists\, x : t\ \phi) \vee (\exists\, x : t.\ \psi)$

Are the following formulas also valid?

- $(\forall\, x : t.\ \phi \vee \psi) \leftrightarrow (\forall\, x : t.\ \phi) \vee (\forall\, x : t.\ \psi)$
  - No! On the left, each x must make either $\phi$ or $\psi$ true. On the right, one of $\phi$ or $\psi$ must hold for every x.
- $(\exists\, x : t.\ \phi \wedge \psi) \leftrightarrow (\exists\, x : t.\ \phi) \wedge (\exists\, x : t.\ \psi)$
  - No! On the left, must pick same x for $\phi$ and $\psi$. On the right, might pick different x for $\phi$ and $\psi$.

# Formal Specification Examples

int[] sort(int[] a)

- requires: $a \neq null$
- ensures: $isSorted(sort(a)) \wedge isPermutationOf(sort(a),a)$

# Formal Specification Examples

int[] sort(int[] a)

- requires: $a \neq null$
- ensures: isSorted(sort(a)) $\wedge$ isPermutationOf(sort(a),a)

int binarySearch(int[] a,int elem)

- requires: $a \neq null \wedge$ isSorted(a)
- ensures:
  (result $=$ -1 $\wedge\ \forall$ i : int, $0 \leq i <$ a.length $\rightarrow$ a[i] $\neq$ elem)
  $\vee$
  (a[result] $=$ elem $\wedge\ \forall$ i : int, $0 \leq i <$ result $\rightarrow$ a[i] $\neq$ elem)

# Today we learned...

- What design by contract is.
- Pre-conditions and post-conditions.
- Formal specification: what and why.
- First-order logic.