

Testing, Debugging, and Verification

Formal Specification, Part II

Srinivas Pinisetty

23 November 2017

Today:

- ▶ Introduction to Dafny:
An imperative language with integrated support for formal specification and verification
- ▶ Methods, assertions, functions and arrays.
- ▶ Classes in Dafny.
- ▶ Pre- and post conditions in Dafny.

Recap: First-Order Logic

Recall: FOL extends propositional logic by:

- ▶ Quantifiers: \forall and \exists .
- ▶ Variables and Types (other than *bool*).
- ▶ (Mathematical) Functions and Predicates (boolean functions)

Example: Σ_{int} :

- ▶ $T_{int} = \{int, bool\}$
- ▶ $F_{int} = \{+, -\} \cup \{\dots, -2, -1, 0, 1, 2, \dots\}$
- ▶ $P_{int} = \{<\}$
- ▶ $\alpha(+)$ = $\alpha(-)$ = $(int, int) \rightarrow int$
 $\alpha(<) = (int, int) \rightarrow bool$
- ▶ In addition, set of (typed) variables V .

Recap: Boolean Connectives and Quantifiers

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

FOL	Meaning	Dafny
$\neg A$	not A	!A
$A \wedge B$	A and B	A && B
$A \vee B$	A or B	A B

Recap: Boolean Connectives and Quantifiers

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

FOL	Meaning	Dafny
$\neg A$	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 ==> B
$A \leftrightarrow B$	A is equivalent of B, A if and only if B	A <==> B

Recap: Boolean Connectives and Quantifiers

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

FOL	Meaning	Dafny
$\neg A$	not A	<code>!A</code>
$A \wedge B$	A and B	<code>A && B</code>
$A \vee B$	A or B	<code>A B</code>
$A \rightarrow B$	A implies B	<code>A ==> B</code>
$A \leftrightarrow B$	A is equivalent of B, A if and only if B	<code>A <==> B</code>
$\forall x : t. A$	For all x of type t , A holds.	<code>forall x:t ::A</code>
$\exists x : t. A$	There exists some x such that A holds.	<code>exists x:t ::A</code>

Example FOL formulas

Example 1: All entries in the array a are greater than 0

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

Example FOL formulas

Example 1: All entries in the array a are greater than 0

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

Example 2: There is at least one prime number in the array a

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

Example FOL formulas

Example 1: All entries in the array a are greater than 0

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

Example 2: There is at least one prime number in the array a

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

Exercise:

Are the following equivalent to the corresponding examples?

Why/why not?

1) $\forall i : int. 0 \leq i < a.Length \wedge a[i] > 0$

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

Example FOL formulas

Example 1: All entries in the array a are greater than 0

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

Example 2: There is at least one prime number in the array a

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

Exercise:

Are the following equivalent to the corresponding examples?

Why/why not?

1) $\forall i : int. 0 \leq i < a.Length \wedge a[i] > 0$

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

No, consider any -ve value.

Example FOL formulas

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

All entries in the array *a* are greater than 0.

- ▶ consider $i = -5$
- ▶ $0 \leq -5 < a.Length \rightarrow a[-5] > 0$
- ▶ $false \rightarrow (P) == true$
- ▶ if -5 was a number between 0 and *arr.Length*, then the element index at -5 would be positive

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

All numbers are between 0 and *arr.Length* and for each number *i* if we access *arr* at that number we get something positive.

- ▶ consider $i = -5$
- ▶ $0 \leq -5 < a.Length \wedge a[-5] > 0$
- ▶ $false \wedge (P) == false$
- ▶ -5 is a number between 0 and *arr.Length* and the element index at -5 would be positive

The Dafny language

Dafny is an imperative language with integrated support for **formal specification and verification**.

Programming language that enforces for each method that if precondition (**Requires**) holds then the postcondition (**Ensure**) must hold

In other words: no breaches of contract! no bugs!

About Dafny

- ▶ Object oriented, similar to Java.
- ▶ Classes, methods.
- ▶ Methods annotated with pre- and post-conditions, loop invariants...
 - ▶ Annotations written in FOL.
- ▶ **Specification automatically checked and proved.**

The Dafny Language

Programming language + Specification language !!

```
method example(a : array<int>)  
  modifies a  
  requires a != null && a.Length > 3  
  requires forall i : int :: 0 <= i < a.Length ==> a[i] > 0  
  ensures forall i : int :: 0 <= i < a.Length && i != 2 ==> a[i] == old(a[i])  
  ensures exists i : int :: 0 <= i < a.Length && a[i] == 42  
  { a[2] := 42; }
```

Programming language

- ▶ Assignments
- ▶ While loops
- ▶ Methods
- ▶ ...

Executed at runtime

Specification language

- ▶ Method pre/post conditions
- ▶ Quantifiers
- ▶ Functions
- ▶ Predicates
- ▶ ...

Used only for checking, ignored at runtime

Classes:

- ▶ Keyword `class`. No access modifiers like `public`, `private` as in Java.
- ▶ Fields declared by `var` keyword (like local variables).
- ▶ Several ways of initialising new objects
 - ▶ Constructors
 - ▶ Initialisation methods

Classes in Dafny

Example: A class in Dafny

```
class MyClass{  
    var x : int; // an integer field  
    constructor(init_x : int){...}  
    method Init(init_x : int){...}  
}
```

Example: Declaring an object

```
// Alt 1: Call anon. constructor  
var myObject := new MyClass(5);  
  
// Alt 2: Using Init-method  
var myObject := new MyClass.Init(5);  
  
//Alt 3: Initialise afterwards.  
var myObject := new MyClass;  
myObject.Init(5);
```

Variables

- ▶ Declared with keyword `var`. Types declared by `:`
- ▶ Assignment written `:=`. Equality check written `==`.
- ▶ Several variables can be declared at once.
- ▶ Parallel assignments possible.

Examples:

```
var x : int;  
x := 34;  
var y, z := true, false;
```


Dafny methods and variables

Like Java, Dafny has methods.

Methods

- ▶ Explicit names for return values.
- ▶ Can refer to return values in specifications.

Example:

```
method ManyReturns(x:int,y:int) returns (more:int,less:int)
)
```

Assertions: The assert keyword

```
method Abs(x : int) returns (r : int)
  ensures 0 <= r;
  {
  if (x < 0) {r := -x;}
  else {r := x;}
  }
method Test(){
  var v := Abs(3);
  assert 0 <= v;
```

- ▶ Placed in the middle of a method.
- ▶ Assertions in Dafny are proven (or rejected) at compile-time, instead of checked at runtime. [Use specification language for what to assert.](#)
- ▶ Dafny tries to prove that assertion holds for all executions of the code.
 - ▶ Easy, follows from postcondition of Abs

The assert keyword

```
method Abs(x : int) returns (r : int)
  ensures 0 <= r;
  {
  if (x < 0) {r := -x;}
  else {r := x;}
  }
method Test(){
  var v := Abs(3);
  assert v == 3;
```

- ▶ Dafny cannot prove this! Why?

The assert keyword

```
method Abs(x : int) returns (r : int)
  ensures 0 <= r;
  {
  if (x < 0) {r := -x;}
  else {r := x;}
  }
method Test(){
  var v := Abs(3);
  assert v == 3;
```

- ▶ Dafny cannot prove this! Why?
- ▶ Dafny **only** remembers the current method body (for efficiency).

The assert keyword

```
method Abs(x : int) returns (r : int)
  ensures 0 <= r;
  {
  if (x < 0) {r := -x;}
  else {r := x;}
  }
method Test(){
  var v := Abs(3);
  assert v == 3;
```

- ▶ Dafny cannot prove this! Why?
- ▶ Dafny **only** remembers the current method body (for efficiency).
- ▶ **Dafny only uses annotations (requires and ensures) of other methods to prove things.**

The assert keyword

```
method Abs(x : int) returns (r : int)
  ensures 0 <= r;
  {
  if (x < 0) {r := -x;}
  else {r := x;}
  }
method Test(){
  var v := Abs(3);
  assert v == 3;
```

- ▶ Dafny cannot prove this! Why?
- ▶ Dafny **only** remembers the current method body (for efficiency).
- ▶ **Dafny only uses annotations (requires and ensures) of other methods to prove things.**
- ▶ Inside Test: Dafny only knows that Abs produce a non-negative result!

Exercise: Fixing the specification of Abs

Revise the specification of Abs in such a way so that Dafny will manage to prove the assertion `assert v == 3`.

```
method Abs(x : int) returns (r : int)
  ensures 0 <= r;
  . . .
method Test(){
  var v := Abs(3);
  assert v == 3; }
```

Exercise: Fixing the specification of Abs

Revise the specification of Abs in such a way so that Dafny will manage to prove the assertion `assert v == 3`.

```
method Abs(x : int) returns (r : int)
  ensures 0 <= r;
  ensures 0 <= x ==> r == x;
  ensures x < 0 ==> r == -x;
  . . .
method Test(){
  var v := Abs(3);
  assert v == 3; }
```

- ▶ Need to specify exactly what the result is in each case.

Dafny Functions

- ▶ Part of specification language (no assignment, while loop).
- ▶ **Cannot modify anything** (unlike methods). Safe to use in spec.
- ▶ **Can only be used in spec (annotations)**.
- ▶ Single unnamed return value, body is single statement (no semicolon).

A function

```
function abs(x : int) : int
  { if x < 0 then -x else x }
```

- ▶ Now, can write e.g. `ensures r == abs(x)`.

A function method

```
function method abs(x : int) : int {  
    if x < 0 then -x else x  
}
```

- ▶ “function method” can be used both from spec and programming !!

Predicates

Functions returning a boolean are called **predicates**.

A predicate

```
predicate ready()  
.....  
{ insertedCard == null && wrongPINCounter == 0 &&  
  auth == false; }
```

Predicates

Functions returning a boolean are called **predicates**.

A predicate

```
predicate ready()  
.....  
{ insertedCard == null && wrongPINCounter == 0 &&  
  auth == false; }
```

Predicates are useful for "naming" common properties used in many annotations:

Example

```
method spitCardOut() returns (card : BankCard)  
.....  
.....  
    ensures ready();
```

Predicates

A predicate is a function giving a boolean

```
predicate isEven(a:int)  
{ a % 2 == 0 }
```

=

```
function isEven(a:int) : bool  
{ a % 2 == 0 }
```

A predicate method is a function method giving a boolean

```
predicate method isEven(a:int)  
{ a % 2 == 0 }
```

=

```
function method isEven(a:int) : bool  
{ a % 2 == 0 }
```

Modifies clauses

```
method enterPIN (pin : int)
  modifies this`auth, this`wrongPINCounter,
  this.insertedCard`valid;
```

- ▶ **Modifies clauses** specifies what fields the method may change.
- ▶ Nothing else can be changed.
- ▶ If a method tries to change something not in its modifies clause, Dafny's compiler complains.
- ▶ Saves writing postconditions for fields that never changes.

Note: Fields of must be prefixed by the ` (backtick) character in modifies clauses.

- ▶ Functions must specify what they read.
- ▶ Methods do not have to specify what they read, only what they modify.
- ▶ We will cover this in the next lecture!

Multiple requires (ensures) Vs using &&

```
method enterPIN (pin : int)
    requires !customerAuth;
    requires insertedCard != null;
    requires insertedCard.valid();
```

specifies the case where **all three** preconditions are true in pre-state

the above is equivalent to:

```
method enterPIN (pin : int)
    requires (!customerAuth && insertedCard != null &&
insertedCard.valid());
```


Parallel for

Declare and initialise an array

```
var a := new int[3];  
a[0], a[1], a[2] := 0, 0, 0;
```

Parallel assignment: Initialise all entries to 0

```
forall(i | 0 <= i < a.Length)  
  {a[i] := 0;}
```

Parallel update: For each index i between 0 and $a.Length$, increment $a[i]$ by 1

```
forall(i | 0 <= i < a.Length)  
  {a[i] := a[i] + 1;}
```

Note: All right-hand side expressions evaluated before assignments.

Some Examples

Let us try some examples !!

Recall: ATM.dfy

```
class ATM {  
  
    // fields:  
    var insertedCard : BankCard;  
    var wrongPINCounter : int;  
    var customerAuth : bool;  
  
    // Initialisation of ATM objects using a  
    constructor:  
    constructor(){...}  
  
    // methods:  
    method insertCard (card : BankCard) { ... }  
    method enterPIN (pin : int) { ... }  
    ...  
  
}
```

Specifying the Init method for class ATM

The Init method is used to initialise new objects:

```
method Init()  
  modifies this;  
  ensures wrongPINCounter == 0;  
  ensures auth == false;  
  ensures insertedCard == null;  
  {  
    insertedCard := null;  
    wrongPINCounter := 0;  
    auth := false;  
  }
```

- ▶ All fields of the object are changed: **modifies this**
- ▶ Postconditions specify initial values.

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 confiscated and blocked. After having entered the correct PIN, the customer is regarded is authenticated.

Implicit: The inserted card is not null. The card is valid to start with (not blocked).

Recall: Specification as Contract

Contract states **what is guaranteed** under which conditions.

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: inserted card is not null, the card is valid.

Dafny by Example

from the file ATM.dfy

```
method enterPIN (pin : int)
```

```
  modifies ...;
```

```
  requires !customerAuth;
```

```
  requires insertedCard != null;
```

```
  requires insertedCard.valid();
```

```
  ensures pin == insertedCard.correctPIN ==> auth;
```

```
  ensures (pin != insertedCard.correctPIN &&  
    wrongPINCounter < 2) ==> . . .
```

```
  ensures (pin != insertedCard.correctPIN &&  
    wrongPINCounter >= 2) ==> . . .
```

Three **pre-conditions** (marked by **requires**)

Three **post-conditions** (marked by **ensures**).

These are *boolean expressions*

Filling in the postconditions...

```
ensures pin == insertedCard.correctPIN ==> auth;  
ensures pin != insertedCard.correctPIN && wrongPINCounter  
  < 2  
=> !auth && wrongPINCounter == old(wrongPINCounter)+1;  
ensures pin != insertedCard.correctPIN && wrongPINCounter  
  >= 2  
    ==> !auth && !insertedCard.valid
```

old(E) means: E evaluated in the pre-state of enterPIN, i.e. the value of E before the method was executed.

Mini Quiz: Specifying insertCard

The informal specification of `insertCard(card:BankCard)` is:

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

A second method `SpitCardOut` is specified simply as:

Returns the bank card currently inserted in the ATM.

Write down a specification for these methods. Recall that the ATM has fields:

```
var insertedCard : BankCard;  
var wrongPINCounter : int;  
var auth : bool;
```

The `BankCard` class has fields

```
var pin : int;  
var accNo : int;  
var valid : bool;
```

```
method insertCard(c : BankCard)
  modifies this`insertedCard;
  requires c != null && c.valid;
  requires this.insertedCard == null && this.auth ==
false && this.wrongPINCounter ==0;
  ensures insertedCard == c;
```

```
method SpitCardOut() returns (card : BankCard)
  modifies this;
  requires insertedCard != null;
  ensures card == old(insertedCard);
  ensures insertedCard == null;
  ensures wrongPINCounter == 0;
  ensures auth == false;
```

Beyond boolean expressions

So far, all annotation has been boolean expressions, e.g. equalities, implications...

How to express the following?

- ▶ An array `arr` only holds values ≤ 2 .

Beyond boolean expressions

So far, all annotation has been boolean expressions, e.g. equalities, implications...

How to express the following?

- ▶ An array `arr` only holds values ≤ 2 .
- ▶ All `BankCard` objects have an account number greater than 0.

Beyond boolean expressions

So far, all annotation has been boolean expressions, e.g. equalities, implications...

How to express the following?

- ▶ An array `arr` only holds values ≤ 2 .
- ▶ All `BankCard` objects have an account number greater than 0.
- ▶ At least one entry in the array is equal to `true`.

Beyond boolean expressions

So far, all annotation has been boolean expressions, e.g. equalities, implications...

How to express the following?

- ▶ An array `arr` only holds values ≤ 2 .
- ▶ All `BankCard` objects have an account number greater than 0.
- ▶ At least one entry in the array is equal to `true`.
- ▶ The variable `m` holds the maximum entry of array `arr`.

Beyond boolean expressions

So far, all annotation has been boolean expressions, e.g. equalities, implications...

How to express the following?

- ▶ An array `arr` only holds values ≤ 2 .
- ▶ All `BankCard` objects have an account number greater than 0.
- ▶ At least one entry in the array is equal to `true`.
- ▶ The variable `m` holds the maximum entry of array `arr`.
- ▶ The array `arr` is sorted between index `i` and `j`.

Beyond boolean expressions

So far, all annotation has been boolean expressions, e.g. equalities, implications...

How to express the following?

- ▶ An array `arr` only holds values ≤ 2 .
- ▶ All `BankCard` objects have an account number greater than 0.
- ▶ At least one entry in the array is equal to `true`.
- ▶ The variable `m` holds the maximum entry of array `arr`.
- ▶ The array `arr` is sorted between index `i` and `j`.

Beyond boolean expressions

So far, all annotation has been boolean expressions, e.g. equalities, implications...

How to express the following?

- ▶ An array `arr` only holds values ≤ 2 .
- ▶ All `BankCard` objects have an account number greater than 0.
- ▶ At least one entry in the array is equal to `true`.
- ▶ The variable `m` holds the maximum entry of array `arr`.
- ▶ The array `arr` is sorted between index `i` and `j`.

Quantifiers!

$\forall x : t. A$	For all x of type t , A holds.	<code>forall x:t :: A</code>
$\exists x : t. A$	There exists some x such that A holds.	<code>exists x:t :: A</code>

Quantifiers: Examples

All BankCard objects have an account number greater than 0

Quantifiers: Examples

All BankCard objects have an account number greater than 0

```
forall b : BankCard :: b.accNo > 0;
```

An array arr only holds only values ≤ 2

Quantifiers: Examples

All BankCard objects have an account number greater than 0

```
forall b : BankCard :: b.accNo > 0;
```

An array arr only holds only values ≤ 2

```
forall i : int :: 0 <= i < arr.Length ==> arr[i] <= 2;
```

At least one entry holds the value true

Quantifiers: Examples

All BankCard objects have an account number greater than 0

```
forall b : BankCard :: b.accNo > 0;
```

An array arr only holds only values ≤ 2

```
forall i : int :: 0 <= i < arr.Length ==> arr[i] <= 2;
```

At least one entry holds the value true

```
exists i : int :: 0 <= i < arr.Length && arr[i] == true;
```

Quantifiers and Range Predicates

- ▶ In this course, most common use of quantifiers in formal specification is to specify properties about arrays (or other data-structures).
- ▶ E.g: Only interested in integers i which are **indices of an array**.
- ▶ **Range predicate**: Restricts range of i more than its type (e.g. not all ints).

Range Predicates

```
forall i : int :: 0 <= i < arr.Length ==> arr[i] <= 2;  
exists i : int :: 0 <= i < arr.Length && arr[i] == true;
```

Today we learned

- ▶ Writing simple classes in Dafny
- ▶ Writing pre- and postconditions for methods.
- ▶ What an assertion is, and a little bit about how Dafny proves them.
- ▶ How to use quantifiers in specifications.

Required Reading: “Getting Started with Dafny: A Guide” (link on course homepage).