

# Formal Methods in Software Engineering

**Specification and Verification** — Spring 2025

Konstantin Chukharev

# §1 Program Verification

## Motivation

Is this program *correct*?

```
x = 0;  
y = a;  
while (y > 0) {  
    x = x + b;  
    y = y - 1;  
}
```

## Program Correctness

**Note:** A program can be *correct* only with respect to a *specification*.

Is this program correct with respect to the following specification? **X**

*“Given integers  $a$  and  $b$ , the program computes and stores in  $x$  the product of  $a$  and  $b$ .”*

## Program Correctness [2]

**Note:** A program can be *correct* only with respect to a *specification*.

Is this program correct with respect to the following specification? ✓

*“Given **positive** integers  $a$  and  $b$ , the program computes and stores in  $x$  the product of  $a$  and  $b$ .”*

```
x = 0;  
y = a;  
while (y > 0) {  
    x = x + b;  
    y = y - 1;  
}
```

## Design by Contract

Specification of a program can be seen as a *contract*:

- *Pre-conditions* define what is *required* to get a meaningful result.
- *Post-conditions* define what is *guaranteed* to return when the precondition is met.

*requires*  $a$  and  $b$  to be positive integers

*ensures*  $x$  is the product of  $a$  and  $b$

## Formal Verification

To formally verify a program you need:

- A formal specification (mathematical description) of the program.
- A formal proof that the specification is correct.
- Automated tools for verification and reasoning.
- Domain-specific expertise.

There are many tools and even specific languages for writing specs and verifying them.

One of them is *Dafny*, both a specification language and a program verifier.

Next, we are going to learn how to:

- *specify* precisely what a program is supposed to do
- *prove* that the specification is correct
- *verify* that the program behaves as specified
- *derive* a program from a specification
- use the *Dafny* programming language and verifier

## §2 Dafny



## Introduction to Dafny

```
method Triple(x: int) returns (r: int)
  ensures r == 3 * x
{
  var y := 2 * x;
  r := x + y;
}
```

**Note:** The *caller* does not need to know anything about the *implementation* of the method, only its *specification*, which abstracts the method's behavior. The method is *opaque* to the caller.

## Introduction to Dafny [2]

Completing the example:

```
method Triple(x: int) returns (r: int)
  requires x >= 0
  ensures r == 3 * x
{
  var y := Double(x);
  r := x + y;
}
```

```
method Double(x: int) returns (r: int)
  requires x >= 0
  ensures r == 2 * x
```

**Exercise:** Fix the above code/spec to avoid `requires x >= 0` in the `Triple` method.

## Logic in Dafny

Dafny expression	Description
true, false	constants
!A	“not $A$ ”
A && B	“ $A$ and $B$ ”
A    B	“ $A$ or $B$ ”
A ==> B	“ $A$ implies $B$ ” or “ $A$ only if $B$ ”
A <==> B	“ $A$ iff $B$ ”
forall x :: A	“for all $x$ , $A$ is true”
exists x :: A	“there exists $x$ such that $A$ is true”

Precedence order: !, &&, ||, ==>, <==>

## Verifying the Imperative Procedure

Below is the Dafny program for computing the maximum segment sum of an array. Source: [1]

```
// find the index range [k..m) that gives the
largest sum of any index range
method MaxSegSum(a: array<int>)
  returns (k: int, m: int)
  ensures  $0 \leq k \leq m \leq a.Length$ 
  ensures forall p, q ::
     $0 \leq p \leq q \leq a.Length \implies$ 
      Sum(a, p, q)  $\leq$  Sum(a, k, m)
{
  k, m := 0, 0;
  var s, n, c, t := 0, 0, 0, 0;
  while n < a.Length
    invariant  $0 \leq k \leq m \leq n \leq a.Length \ \&\&$ 
      s == Sum(a, k, m)
    invariant forall p, q ::
       $0 \leq p \leq q \leq n \implies$  Sum(a, p, q)  $\leq$  s
    invariant  $0 \leq c \leq n \ \&\&$  t == Sum(a, c, n)
    invariant forall b ::
       $0 \leq b \leq n \implies$  Sum(a, b, n)  $\leq$  t
```

```
{
  t, n := t + a[n], n + 1;
  if t < 0 {
    c, t := n, 0;
  } else if s < t {
    k, m, s := c, n, t;
  }
}
}

// sum of the elements in the index range [m..n)
function Sum(a: array<int>, m: int, n: int): int
  requires  $0 \leq m \leq n \leq a.Length$ 
  reads a
{
  if m == n then 0
  else Sum(a, m, n-1) + a[n-1]
}
```

## Program State

```
method MyMethod(x: int) returns (y: int)
  requires x >= 10
  ensures y >= 25
{
  var a := x + 3;
  var b := 12;
  y := a + b;
}
```

The program variables  $x$ ,  $y$ ,  $a$ , and  $b$ , together the method's *state*.

**Note:** Not all program variables are in scope the whole time.

## Floyd Logic

Let's propagate the precondition *forward*:

```
method MyMethod(x: int) returns (y: int)
  requires x >= 10
  ensures y >= 25
{
  // here, we know x >= 10
  var a := x + 3;
  // here, x >= 10 && a == x+3
  var b := 12;
  // here, x >= 10 && a == x+3 && b == 12
  y := a + b;
  // here, x >= 10 && a == x+3 && b == 12 && y == a + b
}
```

The last constructed condition *implies* the required postcondition:

$$(x \geq 10) \wedge (a = x + 3) \wedge (b = 12) \wedge (y = a + b) \rightarrow (y \geq 25)$$

## Floyd Logic [2]

Now, let's go *backward* starting with a postcondition at the last statement:

```
method MyMethod(x: int) returns (y: int)
  requires x >= 10
  ensures y >= 25
{
  // here, we want x + 3 + 12 >= 25
  var a := x + 3;
  // here, we want a + 12 >= 25
  var b := 12;
  // here, we want a + b >= 25
  y := a + b;
  // here, we want y >= 25
}
```

The last calculated condition is *implied* by the given precondition:

$$(x + 3 + 12 \geq 25) \leftarrow (x \geq 10)$$

## Exercise #1

Consider a method with the type signature below which returns in  $s$  the sum of  $x$  and  $y$ , and in  $m$  the maximum of  $x$  and  $y$ :

```
method MaxSum(x: int, y: int)  
  returns (s: int, m: int)  
  ensures ...
```

Write the postcondition specification for this method.



## Exercise #2

Consider a method that attempts to reconstruct the arguments  $x$  and  $y$  from the return values of `MaxSum`. In other words, in other words, consider a method with the following type signature and the same postcondition as in Exercise 1:

```
method ReconstructFromMaxSum(s: int, m: int)
  returns (x: int, y: int)
  requires ...
  ensures ...
```

This method cannot be implemented as is.

Write an appropriate precondition for the method that allows you to implement it.

## §3 Floyd-Hoare Logic

## From Contracts to Floyd-Hoare Logic

In the design-by-contract methodology, contracts are usually assigned to procedures or modules.

In general, it is possible to assign contracts to each statement of a program.

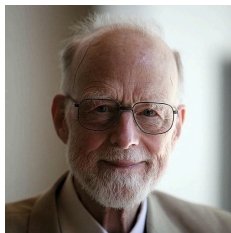
A formal framework for doing this was developed by Tony Hoare, formalizing a reasoning technique by Robert Floyd.

It is based on the notion of a *Hoare triple*.

*Dafny* is based on Floyd-Hoare Logic.



Robert Floyd



Tony Hoare

# Hoare Triples

**Definition 1:** For predicates  $P$  and  $Q$ , and a problem  $S$ , the Hoare triple  $\{P\}S\{Q\}$  describes how the execution of a piece of code changes the state of the computation.

It can be read as “if  $S$  is started in any state that satisfies  $P$ , then  $S$  will terminate (and does not crash) in a state that satisfies  $Q$ ”.

## Examples:

$$\begin{array}{lll} \{x = 1\} & x := 20 & \{x = 20\} \\ \{x < 18\} & y := 18 - x & \{y \geq 0\} \\ \{x < 18\} & y := 5 & \{y \geq 0\} \end{array}$$

## Non-examples:

$$\{x < 18\} \quad x := y \quad \{y \geq 0\}$$

## Forward Reasoning

**Definition 2:** *Forward reasoning* is a construction of a *post-condition* from a given pre-condition.

**Note:** In general, there are *many* possible post-conditions.

**Examples:**

$$\{x = 0\} \quad y := x + 3 \quad \{y < 100\}$$

$$\{x = 0\} \quad y := x + 3 \quad \{x = 0\}$$

$$\{x = 0\} \quad y := x + 3 \quad \{0 \leq x, y = 3\}$$

$$\{x = 0\} \quad y := x + 3 \quad \{3 \leq y\}$$

$$\{x = 0\} \quad y := x + 3 \quad \{\text{true}\}$$

## Strongest Postcondition

Forward reasoning constructs the *strongest* (i.e., *the most specific*) postcondition.

$$\{x = 0\} \quad y := x + 3 \quad \{0 \leq x \wedge y = 3\}$$

**Definition 3:**  $A$  is *stronger* than  $B$  if  $A \rightarrow B$  is a valid formula.

**Definition 4:** A formula is *valid* if it is true for any valuation of its free variables.

## Backward Reasoning

**Definition 5:** *Backward reasoning* is a construction of a *pre-condition* for a given post-condition.

**Note:** Again, there are *many* possible pre-conditions.

**Examples:**

$$\{x \leq 70\} \quad y := x + 3 \quad \{y \leq 80\}$$

$$\{x = 65, y < 21\} \quad y := x + 3 \quad \{y \leq 80\}$$

$$\{x \leq 77\} \quad y := x + 3 \quad \{y \leq 80\}$$

$$\{x \cdot x + y \cdot y \leq 2500\} \quad y := x + 3 \quad \{y \leq 80\}$$

$$\{\text{false}\} \quad y := x + 3 \quad \{y \leq 80\}$$

## Weakest Precondition

Backward reasoning constructs the *weakest* (i.e., *the most general*) pre-condition.

$$\{x \leq 77\} \quad y := x + 3 \quad \{y \leq 80\}$$

**Definition 6:**  $A$  is *weaker* than  $B$  if  $B \rightarrow A$  is a valid formula.



## Weakest Precondition for Assignment

**Definition 7:** The weakest pre-condition for an *assignment* statement  $x := E$  with a post-condition  $Q$ , is constructed by replacing each  $x$  in  $Q$  with  $E$ , denoted  $Q[x := E]$ .

$$\{Q[x := E]\} \quad x := E \quad \{Q\}$$

**Example:** Given a Hoare triple  $\{?\} y := a + b \{25 \leq y\}$ , we construct a pre-condition  $\{25 \leq a + b\}$ .

**Examples:**

$$\{25 \leq x + 3 + 12\} \quad a := x + 3 \quad \{25 \leq a + 12\}$$

$$\{x + 1 \leq y\} \quad x := x + 1 \quad \{x \leq y\}$$

$$\{6x + 5y < 100\} \quad x := 2 \cdot x \quad \{3x + 5y < 100\}$$

## Swap Example

TODO

```
var tmp := x;  
x := y;  
y := tmp;
```

## Simultaneous Assignment

Dafny allows simultaneous assignment of multiple variables in a single statement.

### **Examples:**

$x, y := 3, 10;$                 sets  $x$  to 3 and  $y$  to 10

$x, y = x + y, x - y;$     sets  $x$  to the sum of  $x$  and  $y$  and  $y$  to their difference

All right-hand sides are evaluated *before* any variables are assigned.

**Note:** The last example is *different* from the two statements  $x = x + y; y = x - y;$

## Weakest Precondition for Simultaneous Assignment

**Definition 8:** The weakest pre-condition for a *simultaneous assignment*  $x_1, x_2 := E_1, E_2$  is constructed by replacing each  $x_1$  with  $E_1$  and each  $x_2$  with  $E_2$  in post-condition  $Q$ .

$$Q[x_1 := E_1, x_2 := E_2] \quad x_1, x_2 := E_1, E_2 \quad \{Q\}$$

**Example:**

```
// { x == X, y == Y }  
// { y == Y, x == X }  
x, y = y, x  
// { x == Y, y == X }
```

## Weakest Precondition for Variable Introduction

**Note:** The statement `var x := tmp;` is actually *two* statements: `var x; x := tmp;`

What is true about  $x$  in the post-condition, must have been true for all  $x$  before the variable introduction.

$$\{\forall x. Q\} \quad \text{var } x \quad \{Q\}$$

**Examples:**

- $\{\forall x. 0 \leq x\} \quad \text{var } x \quad \{0 \leq x\}$
- $\{\forall x : \text{int}. 0 \leq x \cdot x\} \quad \text{var } x \quad \{0 \leq x \cdot x\}$

## Strongest Postcondition for Variable Introduction

Consider the Hoare triple  $\{w < x, x < y\} x := 100 \{?\}$ .

Obviously,  $x = 100$  is a post-condition, however it is *not the strongest*.

Something *more* is implied by the pre-condition: there exists an  $n$  such that  $(w < n) \wedge (n < y)$ , which is equivalent to  $w + 1 < y$ .

In general:

$$\{P\} \quad x := E \quad \{\exists n. P[x := n] \wedge x = E[x := n]\}$$

## $\mathcal{WP}$ and $\mathcal{SP}$

Let  $P$  be a predicate on the pre-state of a program  $S$  and let  $Q$  be a predicate on the post-state of  $S$ .

$\mathcal{WP}[S, Q]$  denotes the weakest precondition of  $S$  w.r.t.  $Q$ .

- $\mathcal{WP}[x := E, Q] = Q[x := E]$

$\mathcal{SP}[S, P]$  denotes the strongest postcondition of  $S$  w.r.t.  $P$ .

- $\mathcal{SP}[x := E, P] = \exists n. P[x := n] \wedge x = E[x := n]$

## Control Flow

---

- Assignment:  $x := E$
- Variable introduction:  $\text{var } x$
- Sequential composition:  $S ; T$
- Conditions:  $\text{if } B \{ S \} \text{ else } \{ T \}$
- Method calls:  $r := M(E)$
- Loops:  $\text{while } B \{ S \}$



## Sequential Composition

$S;T$

$\{P\}S\{Q\}T\{R\}$

$\{P\}S\{Q\}$    and    $\{Q\}T\{R\}$

Strongest post-condition:

- Let  $Q = \mathcal{SP}[S, P]$
- $\mathcal{SP}[S;T, P] = \mathcal{SP}[T, Q] = \mathcal{SP}[T, \mathcal{SP}[S, P]]$

Weakest pre-condition:

- Let  $Q = \mathcal{WP}[T, R]$
- $\mathcal{WP}[S;T, R] = \mathcal{WP}[S, Q] = \mathcal{WP}[S, \mathcal{WP}[T, R]]$

# TODO

---



...

## Bibliography

- [1] M. Leino and K. Rustan, “Accessible Software Verification with Dafny,” *IEEE Software*, vol. 34, no. 6, pp. 94–97, Nov. 2017, doi: [10.1109/MS.2017.4121212](https://doi.org/10.1109/MS.2017.4121212).