Program Verification

Readings: Chapter 4 of LICS2

EECS4315 Z:
Mission-Critical Systems
Winter 2025

CHEN-WEI WANG

Learning Objectives

1. Motivating Examples: ***Program Correctness***
2. ***Hoare Triple***
3. ***Weakest Precondition*** (*wp*)
4. Rules of *wp* ***Calculus***
5. Contract of Loops (***invariant*** vs. ***variant***)
6. ***Correctness Proofs*** of Loops

Assertions: Weak vs. Strong

- Describe each assertion as ***a set of satisfying value***.
  $x > 3$ has satisfying values $\{ x \mid x > 3 \} = \{ 4, 5, 6, 7, \dots \}$
  $x > 4$ has satisfying values $\{ x \mid x > 4 \} = \{ 5, 6, 7, \dots \}$
- An assertion p is ***stronger*** than an assertion q if p's set of satisfying values is a subset of q's set of satisfying values.
  - Logically speaking, p being stronger than q (or, q being weaker than p) means $p \Rightarrow q$.
  - e.g., $x > 4 \Rightarrow x > 3$
- What's the weakest assertion? [TRUE]
- What's the strongest assertion? [FALSE]
- In ***System Specification***:
  - A weaker ***invariant*** has more acceptable object states
    e.g., $balance > 0$ vs. $balance > 100$ as an invariant for `ACCOUNT`
  - A weaker ***precondition*** has more acceptable input values
  - A weaker ***postcondition*** has more acceptable output values

Assertions: Preconditions

Given *preconditions* P_1 and P_2, we say that

P_2 ***requires less*** than P_1 if

P_2 is ***less strict*** on (thus ***allowing more***) inputs than P_1 does.

$$\{ x \mid P_1(x) \} \subseteq \{ x \mid P_2(x) \}$$

More concisely:

$$P_1 \Rightarrow P_2$$

e.g., For command `withdraw(amount: INTEGER)`,
$P_2 : amount \geq 0$ ***requires less*** than $P_1 : amount > 0$

What is the *precondition* that ***requires the least***? [***true***]

Assertions: Postconditions

Given *postconditions* or *invariants* Q_1 and Q_2, we say that

Q_2 ***ensures more*** than Q_1 if
Q_2 is ***stricter*** on (thus ***allowing less***) outputs than Q_1 does.

$$\{ x \mid Q_2(x) \} \subseteq \{ x \mid Q_1(x) \}$$

More concisely:

$$Q_2 \Rightarrow Q_1$$

e.g., For query `q(i: INTEGER): BOOLEAN`,

Q_2 : **Result** $= (i > 0) \wedge (i \textbf{ mod } 2 = 0)$ ***ensures more*** than

Q_1 : **Result** $= (i > 0) \vee (i \textbf{ mod } 2 = 0)$

What is the *postcondition* that ***ensures the most***? [***false***]

Motivating Examples (1)

Is this algorithm correct?

```
--algorithm increment_by_9 {
  variable i;
  {
    (* precondition *)
    assert  i > 3

    (* implementation *)
    i := i + 9;

    (* postcondition *)
    assert  i > 13
  }
}
```

Q: Is $i > 3$ is too weak or too strong?
A: Too weak
$\because$ assertion $i > 3$ allows value 4 which would fail postcondition.

Motivating Examples (2)

Is this algorithm correct?

```
--algorithm increment_by_9 {
  variable i;
  {
    (* precondition *)
    assert  i > 5

    (* implementation *)
    i := i + 9;

    (* postcondition *)
    assert  i > 13
  }
}
```

Q: Is $i > 5$ too weak or too strong?
A: Maybe too strong
$\because$ assertion $i > 5$ disallows 5 which would not fail postcondition.
Whether 5 should be allowed depends on the requirements.

Software Correctness

- Correctness is a *relative* notion:
  consistency of *implementation* with respect to *specification*.
  $\Rightarrow$ This assumes there is a specification!
- We introduce a formal and systematic way for formalizing a program **S** and its *specification* (pre-condition ***Q*** and post-condition ***R***) as a *Boolean predicate*: $\{Q\}$ **S** $\{R\}$
  - e.g., $\{i > 3\}$ `i := i + 9` $\{i > 13\}$
  - e.g., $\{i > 5\}$ `i := i + 9` $\{i > 13\}$
  - If $\{Q\}$ **S** $\{R\}$ **can** be proved **TRUE**, then the **S** is correct.
    e.g., $\{i > 5\}$ `i := i + 9` $\{i > 13\}$ can be proved TRUE.
  - If $\{Q\}$ **S** $\{R\}$ **cannot** be proved **TRUE**, then the **S** is incorrect.
    e.g., $\{i > 3\}$ `i := i + 9` $\{i > 13\}$ cannot be proved TRUE.

Hoare Logic

- Consider a program **S** with precondition Q and postcondition R.
  - $\{Q\}$ S $\{R\}$ is a *correctness predicate* for program **S**
  - $\{Q\}$ S $\{R\}$ is TRUE if program **S** starts executing in a state satisfying the precondition Q, and then:
    (a) The program **S** terminates.
    (b) Given that program **S** terminates, then it terminates in a state satisfying the postcondition R.
- Separation of concerns

  (a) requires a proof of *termination*.

  (b) requires a proof of ***partial** correctness*.

  Proofs of (a) + (b) imply ***total** correctness*.

Hoare Logic and Software Correctness

Consider the ***contract***/***specification*** view of an algorithm f (whose body of implementation is **S**) as a Hoare Triple:

$$\{Q\}\ \texttt{S}\ \{R\}$$

Q is the *precondition* of f.
S is the implementation of f.
R is the *postcondition* of f.

- $\{\textit{true}\}$ S $\{R\}$
  All input values are valid [Most-user friendly]
- $\{\textit{false}\}$ S $\{R\}$
  All input values are invalid [Most useless for clients]
- $\{Q\}$ S $\{\textit{true}\}$
  All output values are valid [Most risky for clients; Easiest for suppliers]
- $\{Q\}$ S $\{\textit{false}\}$
  All output values are invalid [Most challenging coding task]
- $\{\textit{true}\}$ S $\{\textit{true}\}$
  All inputs/outputs are valid (No specification) [Least informative]

Proof of Hoare Triple using *wp*

$$\{Q\}\ \texttt{S}\ \{R\} \equiv Q \Rightarrow wp(S, R)$$

- $wp(S, R)$ is the *weakest precondition for S to establish R*.
  - If $Q \Rightarrow wp(S, R)$, then **any** execution started in a state satisfying Q will terminate in a state **satisfying** R.
  - If $Q \not\Rightarrow wp(S, R)$, then **some** execution started in a state satisfying Q will terminate in a state **violating** R.
- S can be:
  - Assignments [`x := y`]
  - Alternations [**if** ... **then** ... **else** ... **end**]
  - Sequential compositions [S_1 ; S_2]
  - Loops [**while** (...) { ... }]
- We will learn how to calculate the *wp* for the above programming constructs.

Denoting Pre- and Post-State Values

- In the *postcondition*, for a program variable x:
  - We write $\boxed{x_0}$ to denote its ***pre-state (old)*** value.
  - We write $\boxed{x}$ to denote its ***post-state (new)*** value.
  Implicitly, in the *precondition*, all program variables have their ***pre-state*** values.
    e.g., $\{b_0 > a\}$ `b := b - a` $\{b = b_0 - a\}$
- Notice that:
  - We may choose to write "b" rather than "b_0" in preconditions
    ∵ All variables are pre-state values in preconditions
  - We don't write "b_0" in program
    ∵ there might be ***multiple intermediate values*** of a variable due to ***sequential composition***

wp Rule: Assignments (1)

$$wp(\texttt{x := e},\ R) = R[x := e]$$

$R[x := e]$ means to substitute all ***free occurrences*** of variable x in postcondition R by expression e.

wp Rule: Assignments (2)

Recall:

$$\{Q\}\ \texttt{S}\ \{R\} \equiv Q \Rightarrow wp(S, R)$$

How do we prove $\{Q\}\ \texttt{x := e}\ \{R\}$?

$$\{Q\}\ \texttt{x := e}\ \{R\} \iff Q \Rightarrow \underbrace{R[x := e]}_{wp(\texttt{x := e},\ R)}$$

wp Rule: Assignments (3) Exercise

What is the weakest precondition for a program `x := x + 1` to establish the postcondition $x > x_0$?

$$\{??\}\ \texttt{x := x + 1}\ \{x > x_0\}$$

For the above Hoare triple to be ***TRUE***, it must be that $?? \Rightarrow wp(\texttt{x := x + 1}, x > x_0)$.

$$
\begin{aligned}
& wp(\texttt{x := x + 1}, x > x_0) \\
= & \ \{ \text{ Rule of } wp\text{: Assignments } \} \\
& x > x_0[x := x_0 + 1] \\
= & \ \{ \text{ Replacing } x \text{ by } x_0 + 1 \ \} \\
& x_0 + 1 > x_0 \\
= & \ \{ \ 1 > 0 \text{ always true } \} \\
& \mathit{True}
\end{aligned}
$$

Any precondition is OK. ***False*** is valid but not useful.

wp Rule: Assignments (4) Exercise

What is the weakest precondition for a program `x := x + 1` to establish the postcondition $x = 23$?

$$\{??\}\ \texttt{x := x + 1}\ \{x = 23\}$$

For the above Hoare triple to be ***TRUE***, it must be that $?? \Rightarrow wp(\texttt{x := x + 1}, x = 23)$.

$$
\begin{aligned}
& wp(\texttt{x := x + 1}, x = 23) \\
= & \ \{ \text{ Rule of } wp\text{: Assignments } \} \\
& x = 23[x := x_0 + 1] \\
= & \ \{ \text{ Replacing } x \text{ by } x_0 + 1 \ \} \\
& x_0 + 1 = 23 \\
= & \ \{ \text{ arithmetic } \} \\
& x_0 = 22
\end{aligned}
$$

Any precondition weaker than $x = 22$ is not OK.

wp Rule: Assignments (4) Revisit

Given $\{??\} n := n + 9 \{n > 13\}$:

- $\boxed{n > 4}$ is the *weakest precondition (wp)* for the given implementation (n := n + 9) to start and establish the postcondition ($n > 13$).
- Any precondition that is ***equal to or stronger than*** the *wp* ($n > 4$) will result in a correct program.

  e.g., $\{n > 5\} n := n + 9 \{n > 13\}$ can be proved **TRUE**.
- Any precondition that is ***weaker than*** the *wp* ($n > 4$) will result in an incorrect program.

  e.g., $\{n > 3\} n := n + 9 \{n > 13\}$ cannot be proved **TRUE**.

  Counterexample: $n = 4$ satisfies precondition $n > 3$ but the output $n = 13$ fails postcondition $n > 13$.

wp Rule: Alternations (1)

$$wp(\texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ end}, R) = \begin{pmatrix} B \Rightarrow wp(S_1, R) \\ \wedge \\ \neg B \Rightarrow wp(S_2, R) \end{pmatrix}$$

The *wp* of an alternation is such that ***all branches*** are able to establish the postcondition ***R***.

wp Rule: Alternations (2)

Recall: $\{Q\}\ S\ \{R\} \equiv Q \Rightarrow wp(S, R)$

How do we prove that $\{Q\}$ **if** B **then** S_1 **else** S_2 **end** $\{R\}$?

```
{Q}
if B then
  {Q ∧ B} S1 {R}
else
  {Q ∧ ¬B} S2 {R}
end
{R}
```

$$\{Q\}\ \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ end}\ \{R\}$$
$$\iff \begin{pmatrix} \{Q \wedge B\}\ S_1\ \{R\} \\ \wedge \\ \{Q \wedge \neg B\}\ S_2\ \{R\} \end{pmatrix} \iff \begin{pmatrix} (Q \wedge B) \Rightarrow wp(S_1, R) \\ \wedge \\ (Q \wedge \neg B) \Rightarrow wp(S_2, R) \end{pmatrix}$$

wp Rule: Alternations (3) Exercise

Is this program correct?

```
{x > 0 ∧ y > 0}
if x > y then
  bigger := x ; smaller := y
else
  bigger := y ; smaller := x
end
{bigger ≥ smaller}
```

$$\begin{pmatrix} \{(x > 0 \wedge y > 0) \wedge (x > y)\} \\ \texttt{bigger := x ; smaller := y} \\ \{bigger \geq smaller\} \end{pmatrix}$$
$$\wedge$$
$$\begin{pmatrix} \{(x > 0 \wedge y > 0) \wedge \neg(x > y)\} \\ \texttt{bigger := y ; smaller := x} \\ \{bigger \geq smaller\} \end{pmatrix}$$

wp Rule: Sequential Composition (1)

$$wp(S_1 \ ; \ S_2, R) = wp(S_1, wp(S_2, R))$$

The *wp* of a sequential composition is such that the first phase establishes the *wp* for the second phase to establish the postcondition ***R***.

wp Rule: Sequential Composition (2)

Recall:

$$\{Q\} \ \mathtt{S} \ \{R\} \equiv Q \Rightarrow wp(S, R)$$

How do we prove $\{Q\} \ S_1 \ ; \ S_2 \ \{R\}$?

$$\{Q\} \ S_1 \ ; \ S_2 \ \{R\} \iff Q \Rightarrow \underbrace{wp(S_1, wp(S_2, R))}_{wp(S_1 \ ; \ S_2, R)}$$

wp Rule: Sequential Composition (3) Exercise

Is $\{ \textit{True} \}$ `tmp := x; x := y; y := tmp` $\{ x > y \}$ correct?

If and only if $\textit{True} \Rightarrow wp($`tmp := x ; x := y ; y := tmp`$, x > y)$

$$
\begin{array}{ll}
 & wp(\mathtt{tmp\ :=\ x\ ;\ \boxed{x\ :=\ y\ ;\ y\ :=\ tmp}}, x > y) \\
= & \{ \ \textit{wp rule for seq. comp.} \ \} \\
 & wp(\mathtt{tmp\ :=\ x}, wp(\mathtt{x\ :=\ y\ ;\ \boxed{y\ :=\ tmp}}, x > y)) \\
= & \{ \ \textit{wp rule for seq. comp.} \ \} \\
 & wp(\mathtt{tmp\ :=\ x}, wp(\mathtt{x\ :=\ y}, wp(\mathtt{y\ :=\ tmp}, x > \boxed{y}))) \\
= & \{ \ \textit{wp rule for assignment} \ \} \\
 & wp(\mathtt{tmp\ :=\ x}, wp(\mathtt{x\ :=\ y}, \boxed{x} > tmp)) \\
= & \{ \ \textit{wp rule for assignment} \ \} \\
 & wp(\mathtt{tmp\ :=\ x}, y > \boxed{\mathrm{tmp}}) \\
= & \{ \ \textit{wp rule for assignment} \ \} \\
 & y > x
\end{array}
$$

$\because$ ***True*** $\Rightarrow y > x$ does not hold in general.

$\therefore$ The above program is not correct.

Loops

- A loop is a way to compute a certain result by ***successive approximations***.

  e.g. computing the maximum value of an array of integers
- Loops are needed and powerful
- But loops ***very hard*** to get right:
  - "off-by-one" error [partial correctness]
  - Not establishing the desired condition [partial correctness]
  - Improper handling of borderline cases [partial correctness]
  - Infinite loops [termination]

Correctness of Loops

How do we prove that the following loop is correct?

```
{Q}
S_init
while(B) {
  S_body
}
{R}
```

In case of C/Java/PlusCal, $\boxed{B}$ denotes the ***stay condition***.

- In TLA+ toolbox, there is **not** native, syntactic support for model-checking the ***total correctness*** of loops.
- Instead, we have to **manually** add assertions to encode:
  - **LOOP INVARIANT** [for establishing ***partial correctness***]
  - **LOOP VARIANT** [for ensuring ***termination***]

Specifying Loops

- Use of ***loop invariant (LI)*** and ***loop variant (LV)***.
  - ***LI***: Boolean expression for measuring/proving ***partial*** *correctness*
    - Typically a special case of the postcondition.
      e.g., Given postcondition "*Result is maximum of the array*":
      LI can be "*Result is maximum of the* ***part of array scanned so far***".
    - ***Established*** before the very first iteration.
    - ***Maintained*** TRUE after each iteration.
  - ***LV***: Integer expression for measuring/proving ***termination***
    - Denotes the "number of iterations remaining"
    - *Decreased* at the end of each subsequent iteration
    - Maintained *non-negative* at the end of each iteration.
    - As soon as value of ***LV*** reaches *zero*, meaning that no more iterations remaining, the loop must exit.
- Remember:

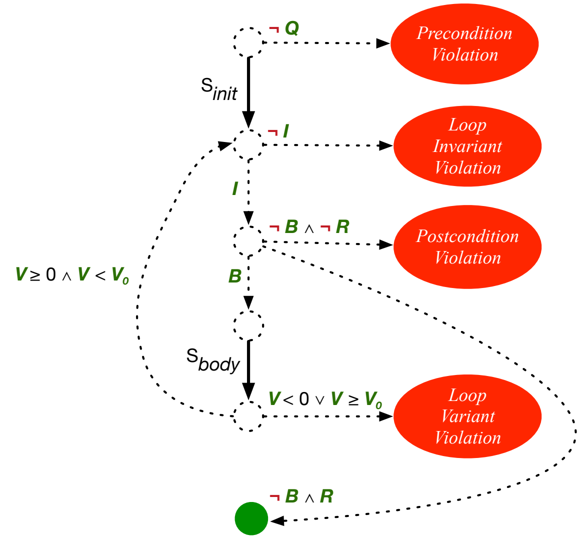
  total** correctness* = ***partial** correctness* + ***termination

Specifying Loops: Syntax

```
CONSTANT ... (* input list *)
I(var_list) == ...
V(var_list) == ...
--algorithm MyAlgorithm {
 variables ..., variant_pre = 0, variant_post = 0;
 {
  assert Q; (* Precondition *)
  S_init
  assert I(...); (* Is LI established? *)
  while( B ) {
   variant_pre := V(...);
   S_body
   variant_post := V(...);

   assert variant_post >= 0;
   assert variant_post < variant_pre;
   assert I(...); (* Is LI preserved? *)
  }
  assert R; (* Postcondition *)
 }
}
```


Specifying Loops: Runtime Checks (1)



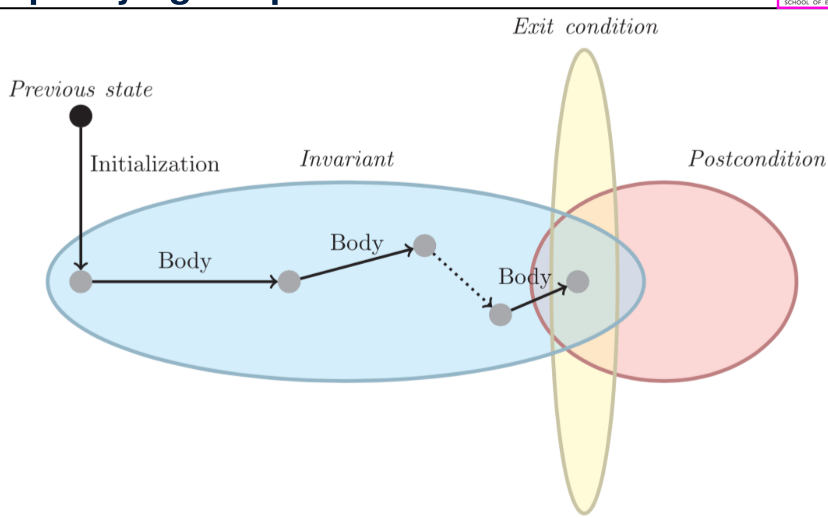
Specifying Loops: Runtime Checks (2)

```
1   I(i) == (1 <= i) /\ (i <= 6)
2   V(i) == 6 - i
3   --algorithm loop_invariant_test
4    variables i = 1, variant_pre = 0, variant_post = 0;
5    {
6     assert I(i);
7     while (i <= 5) {
8       variant_pre := V(i);
9       i := i + 1;
10      variant_post := V(i);
11      assert variant_post >= 0;
12      assert variant_post < variant_pre;
13      assert I(i);
14    } ;
15   }
```

L1: Change to `1 <= i /\ i <= 5` for a ***Loop Invariant Violation***.

L2: Change to `5 - i` for a ***Loop Variant Violation***.

Specifying Loops: Visualization



Digram Source: page 5 in *Loop Invariants: Analysis, Classification, and Examples*

Proving Correctness of Loops (1)

```
{Q}
S_init
assert I(...);
while( B ) {
  variant_pre := V(...);
  S_body
  variant_post := V(...);
  assert variant_post >= 0;
  assert variant_post < variant_pre;
  assert I(...);
}
{R}
```

- A loop is ***partially** correct* if:
  - Given precondition ***Q***, the initialization step S_{init} establishes ***LI*** I.
  - At the end of S_{body}, if not yet to exit, ***LI*** I is maintained.
  - If ready to exit and ***LI*** I maintained, postcondition ***R*** is established.
- A loop ***terminates*** if:
  - Given ***LI*** I, and not yet to exit, S_{body} maintains ***LV*** V as non-negative.
  - Given ***LI*** I, and not yet to exit, S_{body} decrements ***LV*** V.

Proving Correctness of Loops (2)

- A loop is ***partially** correct* if:
  - Given precondition ***Q***, the initialization step S_{init} establishes ***LI*** I.

    $\{Q\}\ S_{init}\ \{I\}$
  - At the end of S_{body}, if not yet to exit, ***LI*** I is maintained.

    $\{I \wedge B\}\ S_{body}\ \{I\}$
  - If ready to exit and ***LI*** I maintained, postcondition ***R*** is established.

    $I \wedge \neg B \Rightarrow R$
- A loop ***terminates*** if:
  - Given ***LI*** I, and not yet to exit, S_{body} maintains ***LV*** V as non-negative.

    $\{I \wedge B\}\ S_{body}\ \{V \geq 0\}$
  - Given ***LI*** I, and not yet to exit, S_{body} decrements ***LV*** V.

    $\{I \wedge B\}\ S_{body}\ \{V < V_0\}$

Index (1)

Index (2)

Index (3)