

Verification conditions and loop invariants

From Hoare rules to loop proofs

CMPUT 664 • Formal Verification

Today

1. From Hoare proofs to verification conditions
2. Weakest preconditions for loop-free commands
3. Assertions and assumptions
4. Annotated loops and their proof obligations
5. Designing loop invariants

From proof rules to a calculation

Last lecture, a Hoare proof followed the structure of a command.

For example, proving

$$\{P\} x := x + 1; y := 2x \{Q\}$$

requires finding an intermediate assertion and applying the assignment rule twice.

Can those assertions and logical side conditions be generated mechanically?

Two parts of deductive verification

Given a program and a pre/postcondition specification:

1. **Verification-condition generation** translates program structure into logical formulas.
2. **A decision procedure or theorem prover** checks whether those formulas are valid.

The first part is syntax-directed. The second depends on the assertion logic.

What is a verification condition?

A verification condition is a logical formula whose validity is sufficient to establish the desired program property.

For

$$\{P\} S \{Q\},$$

the formula must connect the supplied precondition P , the behavior of S , and the required postcondition Q .

We begin with commands that contain no loops.

Weakest preconditions

Reasoning backward from the desired result

A predicate transformer

For a loop-free command S and postcondition Q ,

$$\text{wp}(S, Q)$$

is an assertion describing exactly the initial states from which executing S is guaranteed to terminate in a state satisfying Q .

It transforms a predicate on final states into a predicate on initial states.

Why is it the weakest?

$\text{wp}(S, Q)$ satisfies two properties:

$$\models \{\text{wp}(S, Q)\} S \{Q\},$$

and, for every assertion R ,

$$\models \{R\} S \{Q\} \quad \Rightarrow \quad \models R \Rightarrow \text{wp}(S, Q).$$

Any other sufficient precondition restricts the initial states at least as much.

The verification condition

For a loop-free command S ,

$$\models \{P\} S \{Q\}$$

exactly when

$$\models P \Rightarrow \text{wp}(S, Q).$$

The program calculation produces $\text{wp}(S, Q)$; the remaining question is validity in the assertion logic.

Skip and assignment

$$\text{wp}(\mathbf{skip}, Q) = Q$$

$$\text{wp}(x := E, Q) = Q[E/x]$$

These are the same backward steps used by the Hoare rules.

Sequential composition

For

$$S_1; S_2,$$

first determine what must hold before S_2 , then use that result as the postcondition for S_1 :

$$\text{wp}(S_1; S_2, Q) = \text{wp}(S_1, \text{wp}(S_2, Q)).$$

The intermediate assertion is calculated rather than guessed.

Example: a straight-line program

Determine the initial states that guarantee $y > 10$ after

```
1 x := x + 1;  
2 y := 2 * x
```

We work backward from the required postcondition.

The final assignment

Substitute $2x$ for y in the postcondition:

$$\begin{aligned}\text{wp}(y := 2x, y > 10) &= (y > 10)[2x/y] \\ &= 2x > 10.\end{aligned}$$

This is the required intermediate assertion.

The first assignment

Now substitute $x + 1$ for x :

$$\begin{aligned}\text{wp}(x := x + 1, 2x > 10) &= 2(x + 1) > 10 \\ &\equiv x > 4.\end{aligned}$$

The whole program establishes $y > 10$ from exactly the states satisfying $x > 4$.

Adding a supplied precondition

Suppose the desired triple is

$$\{x \geq 5\} x := x + 1; y := 2x \{y > 10\}.$$

The generated verification condition is

$$x \geq 5 \Rightarrow x > 4.$$

No program syntax remains; this is an integer-arithmetic validity question.

Conditionals

Each possible branch must establish the postcondition:

$$\begin{aligned} & \text{wp}(\mathbf{if } B \mathbf{ then } S_1 \mathbf{ else } S_2, Q) \\ &= (B \Rightarrow \text{wp}(S_1, Q)) \wedge (\neg B \Rightarrow \text{wp}(S_2, Q)). \end{aligned}$$

The guard determines which branch precondition is required.

Example: absolute value

```
1  if x >= 0 then
2      y := x
3  else
4      y := -x
```

Required postcondition:

$$y \geq 0.$$

What precondition does the calculation produce?

Calculate the branch requirements

For the then branch:

$$\text{wp}(y := x, y \geq 0) = x \geq 0.$$

For the else branch:

$$\text{wp}(y := -x, y \geq 0) = -x \geq 0.$$

Combine the branches

The weakest precondition is

$$(x \geq 0 \Rightarrow x \geq 0) \wedge (x < 0 \Rightarrow -x \geq 0).$$

This formula is valid, so the command establishes $y \geq 0$ from every initial state:

$$\{\text{true}\} S \{y \geq 0\}.$$

An alternative direction

Backward reasoning computes weakest preconditions from Q .

Forward reasoning can instead compute a **strongest postcondition** from P :

$$\text{sp}(S, P) \Rightarrow Q.$$

Both can generate verification conditions. We use weakest preconditions because they align directly with substitution and the proof rules from last lecture.

Assertions and assumptions

Obligations and hypotheses inside a program

Two different annotations

```
1 assert R
```

requires the verifier to prove R at that point.

```
1 assume R
```

restricts subsequent reasoning to executions for which R holds.

They therefore have different weakest preconditions.

Weakest precondition for **assert**

To continue with Q , both the assertion and Q must hold:

$$\text{wp}(\text{assert } R, Q) = R \wedge Q.$$

For example,

$$\text{wp}(\text{assert } x > 0, y = x) = x > 0 \wedge y = x.$$

Weakest precondition for **assume**

Executions violating the assumption are excluded:

$$\text{wp}(\text{assume } R, Q) = R \Rightarrow Q.$$

For example,

$$\text{wp}(\text{assume } x > 0, y = x) = x > 0 \Rightarrow y = x.$$

An assumption supplies information; it does not prove that the information is true.

Compare their effect

Starting from true:

```
1 assert  $x > 0$ 
```

cannot be verified, because $x > 0$ is not valid in every state.

```
1 assume  $x > 0$ 
```

is permitted, but all later conclusions apply only under that assumption.

Loops

Where exact backward calculation stops

Why a loop is different

For

```
1 while B do S
```

the effect may require reasoning about arbitrarily many executions of S .

There is no finite syntax-directed expansion analogous to a fixed sequence of assignments.

We therefore require a supplied loop invariant I as a summary.

Hoare rule for while loops

$$\frac{\vdash \{I \wedge B\} S \{I\}}{\vdash \{I\} \mathbf{while} B \mathbf{do} S \{I \wedge \neg B\}} \text{ While}$$

The invariant I holds at the loop head. The premise proves that one iteration preserves it when the guard B holds. At exit, $I \wedge \neg B$ holds.

This rule establishes partial correctness.

An annotated loop

Write

```
1 while B invariant I do S
```

The generator uses the supplied I as its loop summary:

$$\text{awp}(\text{while } B \text{ invariant } I \text{ do } S, Q) = I.$$

This need not be the exact weakest precondition. In the count-up loop below, every integer start eventually satisfies $Q \equiv x \geq n$, but choosing $I \equiv x \leq n$ makes $\text{awp} = x \leq n$ and excludes safe starts with $x > n$.

The next slide shows how I can also include loop-head states that are never reached.

An invariant can include unreachable states

```
1 x := 0;  
2 while x < 10 do x := x + 2
```

The values reached at the loop head are 0, 2, 4, 6, 8, 10.

$I \equiv x \geq 0$ holds at the start and remains true after each iteration. It also admits states such as $x = 1$ and $x = 11$ that this program never reaches.

The invariant is an over-approximation of the reachable loop-head states.

Three obligations for a loop

To verify

$\{P\}$ **while** B **invariant** I **do** S $\{Q\}$,

generate:

1. **Initiation:** $P \Rightarrow I$
2. **Preservation:** $I \wedge B \Rightarrow \text{awp}(S, I)$
3. **Exit:** $I \wedge \neg B \Rightarrow Q$

Preservation is the premise of the while rule. Initiation and exit adapt that rule to the supplied P and Q using consequence.

Example: count up to n

```
 $\{x \leq n\}$   
while  $x < n$  invariant  $x \leq n$  do  
     $x := x + 1$   
 $\{x \geq n\}$ 
```

Here P and I are both $x \leq n$, while Q is $x \geq n$.

Initiation

The invariant must hold on entry:

$$x \leq n \Rightarrow x \leq n.$$

This obligation is valid immediately.

Preservation

The loop body must re-establish the invariant:

$$x \leq n \wedge x < n \Rightarrow \text{wp}(x := x + 1, x \leq n).$$

After substitution:

$$x \leq n \wedge x < n \Rightarrow x + 1 \leq n.$$

This is valid when x and n are integers.

Exit

The invariant and a false guard must establish the postcondition:

$$x \leq n \wedge \neg(x < n) \Rightarrow x \geq n.$$

This obligation is also valid. Together, the three formulas prove partial correctness of the annotated loop.

What has been automated?

The generator can:

- Traverse assignments, sequences, and branches
- Propagate postconditions backward
- Place loop invariants at the appropriate program points
- Produce logical formulas for a solver

It does not, by itself, determine a useful invariant.

If a verification condition is invalid

A model of the negated condition gives a state that violates that logical obligation.

It may reveal:

- An incorrect program or specification
- An invariant that is too weak or not preserved
- A missing assumption

Soundness of VC generation

The essential guarantee is:

$$\text{all generated VCs are valid} \quad \Rightarrow \quad \models \{P\} S \{Q\}.$$

For loop-free commands, weakest-precondition calculation gives an exact characterization. With annotated loops, verification can fail because the supplied invariant is inadequate even when the program is correct. We now examine how to choose and check that invariant.

Designing loop invariants

Summaries of repeated computation

The missing input to verification

For loop-free code, weakest preconditions calculate the necessary intermediate assertions.

For a loop, the verifier requires a finite summary of all possible loop iterations:

```
1 while B invariant I do S
```

How do we choose I ?

The loop head

The invariant describes states at one particular program point: the loop head.

That point is reached:

- Before the first guard evaluation
- Again after every completed iteration
- Once more when the guard becomes false

The same assertion I must hold at each visit.

Two roles of an invariant

An invariant must be:

1. **Inductive:** established initially and preserved by one arbitrary iteration.
2. **Sufficient:** strong enough, together with loop exit, to establish the postcondition.

Invariants as induction

The proof shape is mathematical induction over the number of completed iterations:

- **Base case:** initiation establishes I before zero iterations.
- **Inductive step:** preservation carries I through one more iteration.

After any finite number of iterations, I holds at the loop head.

The exit condition then connects this fact to Q .

Invariant strength

Enough information, but still inductive

Stronger and weaker assertions

If

$$I_1 \Rightarrow I_2,$$

then I_1 is stronger: it describes fewer states.

Strength has competing effects:

- A stronger invariant provides more information at exit.
- A stronger invariant may be harder to establish or preserve.
- A weaker invariant is easier to satisfy but may not prove Q .

Two extreme candidates

$I \equiv \text{true}$ is always established and preserved, but often proves too little at exit.

$I \equiv \text{false}$ makes preservation and exit trivial, but cannot be established from a satisfiable precondition.

Neither extreme generally gives a useful proof.

Is the postcondition an invariant?

The postcondition describes what must hold after the loop.

It may be false before the first iteration or during intermediate iterations, so choosing

$$I = Q$$

usually fails initiation or preservation.

Instead, the postcondition often suggests a relationship that can be generalized to partial progress.

Diagnosing a candidate

Each failed obligation suggests a different revision:

Failed condition	Likely issue
$P \Rightarrow I$	Candidate excludes an initial state
$I \wedge B \Rightarrow \text{wp}(S, I)$	Candidate is not closed under an iteration
$I \wedge \neg B \Rightarrow Q$	Candidate has lost information needed at exit

A countermodel identifies values witnessing the failure.

Sources of invariant information

Useful candidates commonly combine:

- A relation between accumulated values and a progress variable
- Bounds implied by initialization and the guard
- Facts unchanged by the loop
- A postcondition generalized from the full input to the processed prefix

These are heuristics; the proof obligations decide whether the result is adequate.

Case study: summation

Deriving an invariant from the desired result

The program and specification

For an integer $n \geq 0$, compute $1 + 2 + \cdots + n$:

```
1 i := 0;  
2 s := 0;  
3 while i < n do  
4     i := i + 1;  
5     s := s + i
```

Required postcondition:

$$2s = n(n + 1).$$

What changes during the loop?

At exit, the desired equation relates the result s to the input n .

During execution:

- n is fixed
- i records how far the loop has progressed
- s contains the sum accumulated through i

The postcondition describes completed progress; an invariant must also describe partial progress.

Generalize from n to i

Replace the completed range endpoint n by the current endpoint i :

$$I_{\text{sum}} : \quad 2s = i(i + 1).$$

This candidate records the functional relationship between the accumulator and the portion already processed.

Is it inductive? Is it sufficient at exit?

Check initiation

After initialization, $i = 0$ and $s = 0$:

$$2s = i(i + 1)$$

becomes

$$0 = 0(0 + 1).$$

The relationship is established.

Check preservation

Assume before an iteration that

$$2s = i(i + 1).$$

After the body:

$$i' = i + 1, \quad s' = s + i' = s + i + 1.$$

We must show $2s' = i'(i' + 1)$.

Preservation calculation

Using the old-state invariant:

$$\begin{aligned} 2s' &= 2(s + i + 1) \\ &= i(i + 1) + 2(i + 1) \\ &= (i + 1)(i + 2) \\ &= i'(i' + 1). \end{aligned}$$

The generalized relationship is preserved.

Check the exit condition

At exit, the negated guard gives

$$i \geq n.$$

Together with

$$2s = i(i + 1),$$

this does not imply $2s = n(n + 1)$: i might be larger than n .

The candidate is inductive but not yet sufficient.

Recover the missing bound

The loop starts with $i = 0$ and increments i only while $i < n$.

Add the progress bounds:

$$I : \quad 0 \leq i \leq n \wedge 2s = i(i + 1).$$

Now $i \leq n$ and the exit fact $i \geq n$ imply $i = n$.

Recheck initiation

From $n \geq 0$, $i = 0$, and $s = 0$:

$$0 \leq 0 \leq n \quad \wedge \quad 2(0) = 0(0 + 1).$$

The strengthened invariant still holds initially.

Recheck preservation of the bounds

Assume

$$0 \leq i \leq n \wedge i < n.$$

Because $i, n \in \mathbb{Z}$,

$$0 \leq i + 1 \leq n.$$

The functional relationship was already shown to be preserved, so the entire conjunction is inductive.

Establish the postcondition

At loop exit:

$$0 \leq i \leq n \quad \wedge \quad i \geq n$$

implies $i = n$. Substituting into the invariant gives

$$2s = i(i + 1) = n(n + 1).$$

The strengthened invariant satisfies all three obligations.

What the derivation used

The final invariant combines two kinds of information:

$$\underbrace{2s = i(i + 1)}_{\text{partial result}} \quad \wedge \quad \underbrace{0 \leq i \leq n}_{\text{index bounds}} .$$

The first part tracks the computation. The second connects loop exit to completion.

Why the summation loop stops

If it stops: the invariant gives $2s = n(n + 1)$.

Why that is not enough: `while i < n do skip` keeps the invariant true but never stops from a state with $i < n$.

What forces an exit here: the actual loop increments i while n stays fixed. The remaining distance $V = n - i$ is a positive integer while $i < n$, and each iteration reduces it by one:

$$V' = n - (i + 1) = V - 1.$$

A positive integer cannot decrease forever. So this loop stops, and the invariant's exit proof applies.

Quantified invariants

Generalizing a property over an array prefix

Example: maximum of an array

For a nonempty array a :

```
1 i := 1;  
2 m := a[0];  
3 while i < len(a) do  
4     if a[i] > m then m := a[i];  
5     i := i + 1
```

At exit, m should be a maximum element of the whole array.

The postcondition has two parts

m is at least every array value:

$$\forall k. 0 \leq k < \text{len}(a) \Rightarrow a[k] \leq m.$$

m is an element of the array:

$$\exists k. 0 \leq k < \text{len}(a) \wedge m = a[k].$$

The second part prevents an arbitrary upper bound from being accepted as the maximum element.

Generalize the full array to a prefix

At the loop head, indices below i have been processed. A candidate invariant is

$$\begin{aligned} 1 \leq i \leq \text{len}(a), \\ \forall k. 0 \leq k < i \Rightarrow a[k] \leq m, \\ \exists k. 0 \leq k < i \wedge m = a[k]. \end{aligned}$$

Replacing the full range by the processed prefix makes the property meaningful before termination.

Why the candidate is preserved

One new element, $a[i]$, is added to the processed prefix.

- If $a[i] > m$, assigning $m := a[i]$ supplies a new maximum and witness.
- Otherwise, the old m bounds the new element and retains its old witness.
- Incrementing i extends the quantified range by exactly that element.

This argument corresponds to the two branches of the preservation VC.

Why the candidate is sufficient

At exit,

$$i \geq \text{len}(a).$$

The invariant also gives $i \leq \text{len}(a)$, so

$$i = \text{len}(a).$$

The prefix properties therefore become the two required properties over the entire array.

Frames and unchanged data

The argument assumes that the loop does not modify a or its length.

For programs with heap updates, an invariant may also need to describe:

- Which locations may change
- Which values remain equal to their initial values
- Separation or aliasing relationships

A practical invariant workflow

1. Identify the loop head and the meaning of each changing variable.
2. Generalize the postcondition from completed work to partial progress.
3. Add bounds and facts needed to connect a false guard to completion.
4. Check initiation, preservation, and exit separately.
5. Use a failed VC to strengthen or correct the candidate.

Next: Dafny

We now have the main ingredients of deductive verification:

- Pre/postcondition specifications
- Syntax-directed verification-condition generation
- Loop invariants for repeated computation

Next, we will see how these ideas appear in a program verifier: Dafny.