

Computer-Aided Reasoning for Software

# Reasoning about Programs II

**Emina Torlak**

[emina@cs.washington.edu](mailto:emina@cs.washington.edu)

# Overview

## Last two lectures

- Reasoning about (partial) correctness with Hoare Logic
- Reasoning about total correctness with Dafny

## Today

- Automating Hoare Logic with verification condition generation—how tools like Dafny work

# Recap: Imperative Programming Language (IMP)

## Expression E

- $Z \mid V \mid E_1 + E_2 \mid E_1 * E_2$

## Conditional C

- $\text{true} \mid \text{false} \mid E_1 = E_2 \mid E_1 \leq E_2$

## Statement S

- **skip** (Skip)
- **abort** (Abort)
- $V := E$  (Assignment)
- $S_1; S_2$  (Composition)
- **if C then S<sub>1</sub> else S<sub>2</sub>** (If)
- **while C do S** (While)

# Recap: inference rules for Hoare logic

---

$$\vdash \{P\} \text{ skip } \{P\}$$

---

$$\vdash \{\text{true}\} \text{ abort } \{\text{false}\}$$

---

$$\vdash \{Q[E/x]\} x := E \{Q\}$$

---

$$\frac{\vdash \{P_1\} S \{Q_1\} \quad P \Rightarrow P_1 \quad Q_1 \Rightarrow Q}{\vdash \{P\} S \{Q\}}$$

---

$$\frac{\vdash \{P\} S_1 \{R\} \quad \vdash \{R\} S_2 \{Q\}}{\vdash \{P\} S_1; S_2 \{Q\}}$$

---

$$\frac{\vdash \{P \wedge C\} S_1 \{Q\} \quad \vdash \{P \wedge \neg C\} S_2 \{Q\}}{\vdash \{P\} \text{ if } C \text{ then } S_1 \text{ else } S_2 \{Q\}}$$

---

$$\frac{\vdash \{P \wedge C\} S \{P\}}{\vdash \{P\} \text{ while } C \text{ do } S \{P \wedge \neg C\}}$$

*loop invariant*

# Challenge: manual proof construction is tedious!

```
{x ≤ n}
while (x < n) do
  {x ≤ n ∧ x < n}
  {x+1 ≤ n}           // consequence
  x := x + 1         // assignment
  {x ≤ n}           // while
{x ≤ n ∧ x ≥ n}
{x = n}             // consequence
```

Hoare Logic proofs are highly manual:

- When to apply the rule of consequence?
- What loop invariants to use?

# Challenge: manual proof construction is tedious!

```
{x ≤ n} // precondition
while (x < n) do
  {x ≤ n} // loop invariant
  x := x + 1
{x = n} // postcondition
```

Hoare Logic proofs are highly manual:

- When to apply the rule of consequence?
- What loop invariants to use?

We can automate much of the proof process with verification condition generation!

- But loop invariants still need to be provided ...

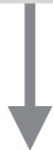
# Automating Hoare logic with VC generation

Program annotated with  
pre/post conditions, loop  
invariants

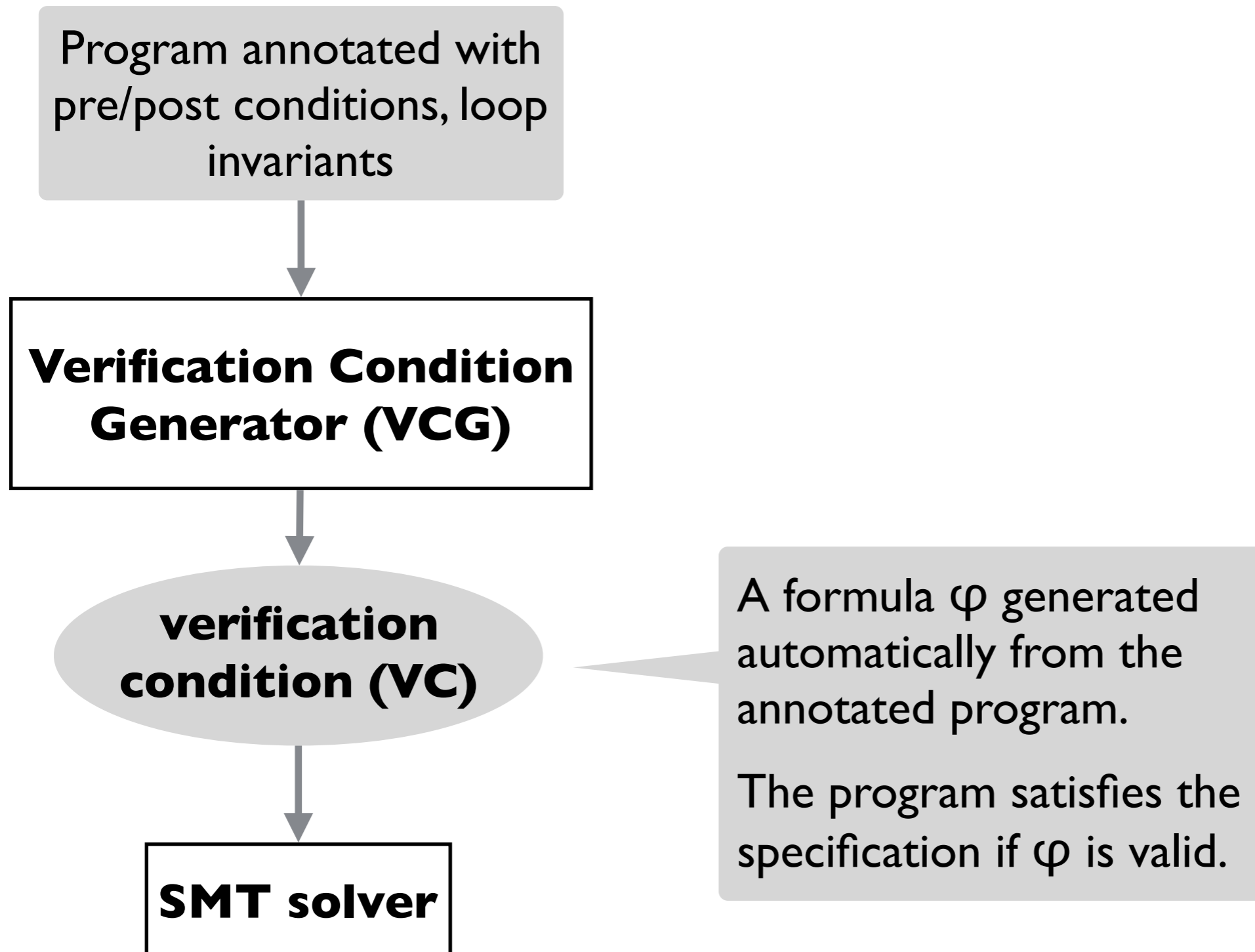
**Verification Condition  
Generator (VCG)**

**verification  
condition (VC)**

**SMT solver**

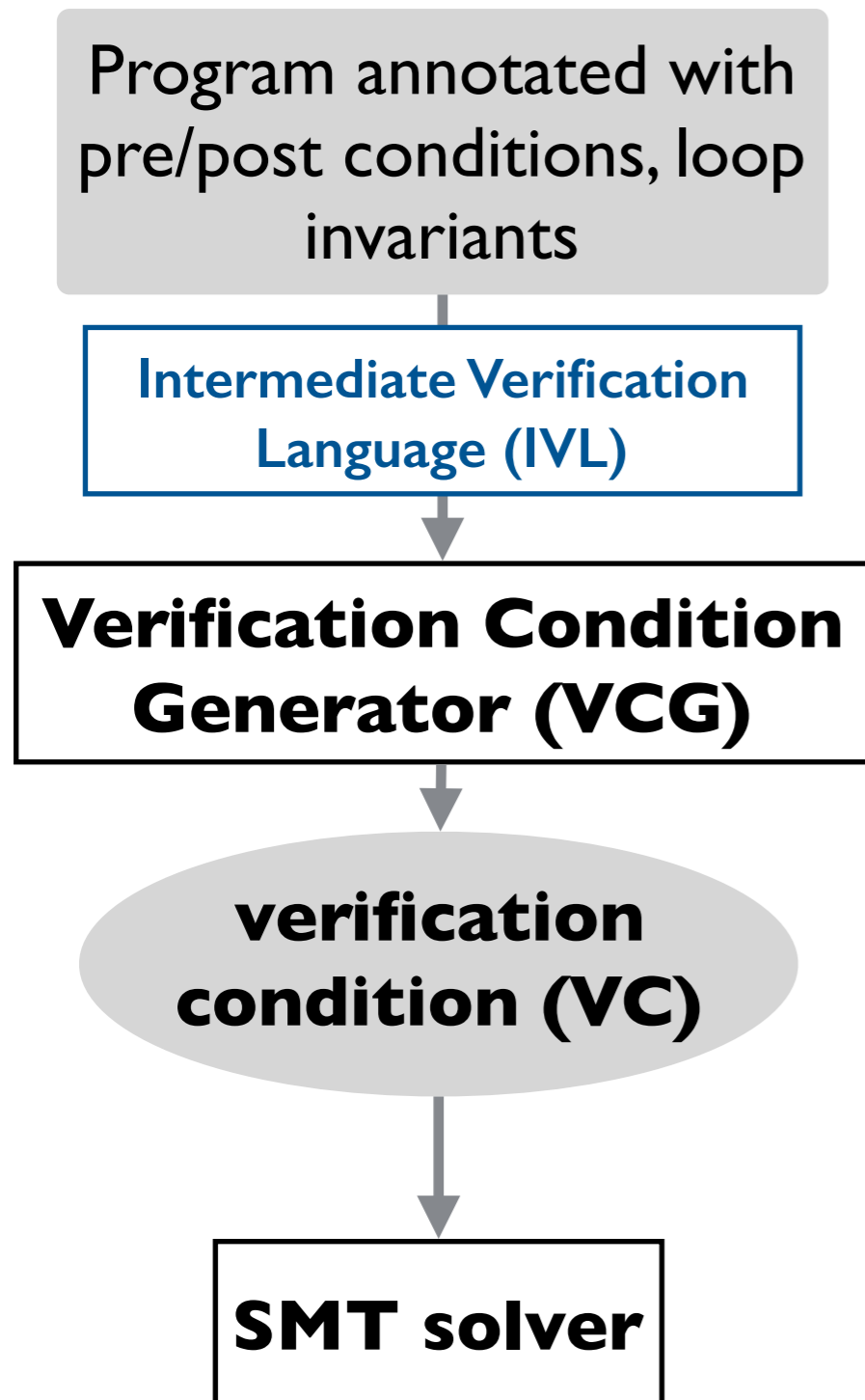


# Automating Hoare logic with VC generation

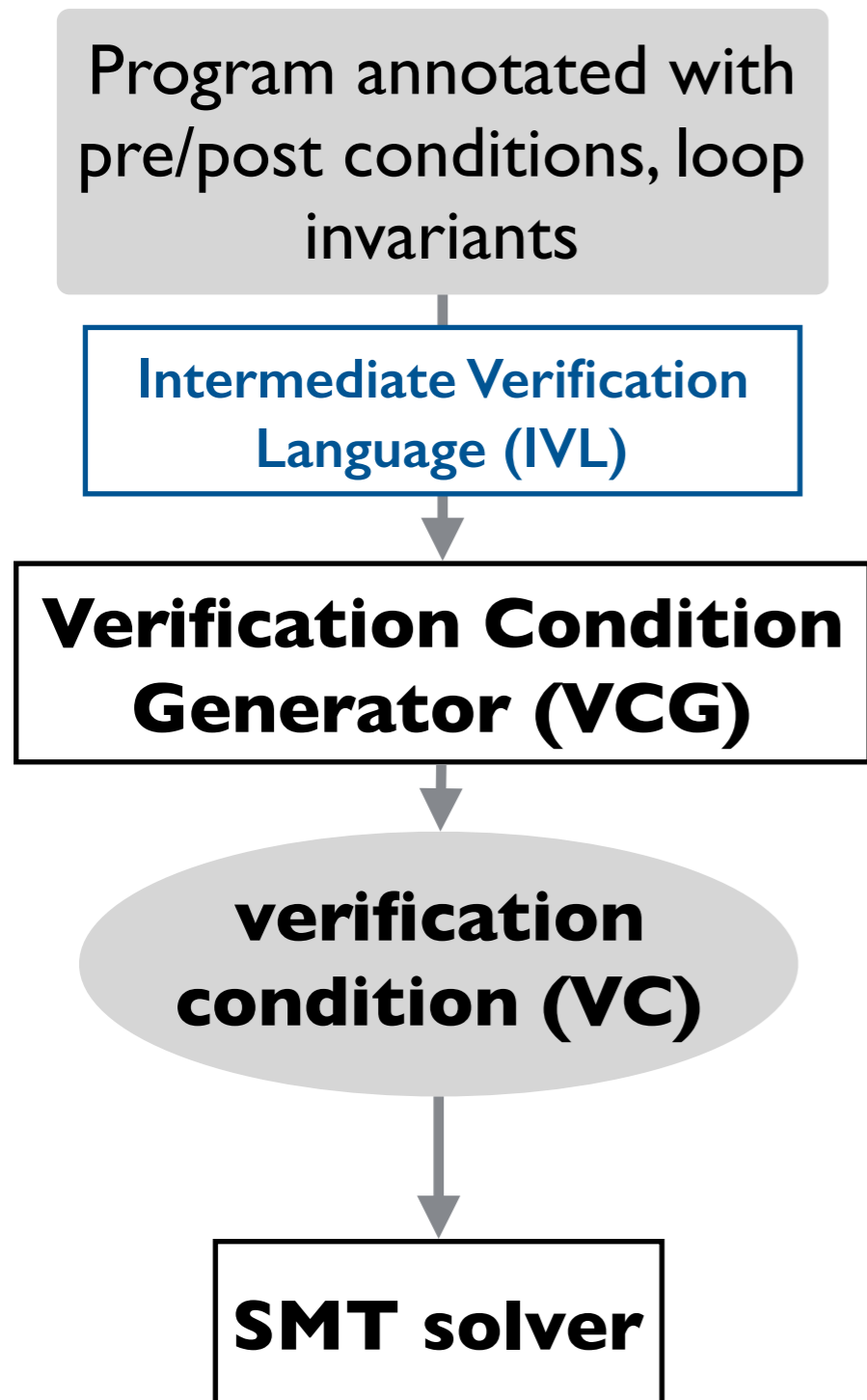




# Automating Hoare logic with VC generation



# Automating Hoare logic with VC generation



## Forwards computation:

- Starting from the precondition, generate formulas to prove the postcondition.
- Based on computing *strongest postconditions (sp)*.

## Backwards computation:

- Starting from the postcondition, generate formulas to prove the precondition.
- Based on computing *weakest liberal preconditions (wp)*.

# **VC generation with WP and SP**

# VC generation with WP and SP

## $sp(S, P)$

- The strongest predicate that holds for states produced by executing  $S$  on a state satisfying  $P$ .

Symbolic execution, covered in next lecture, computes SPs for finite programs (no unbounded loops).

# VC generation with WP and SP

## $sp(S, P)$

- The strongest predicate that holds for states produced by executing  $S$  on a state satisfying  $P$ .

Symbolic execution, covered in next lecture, computes SPs for finite programs (no unbounded loops).

## $wp(S, Q)$

- The weakest predicate that guarantees  $Q$  will hold for states produced by executing  $S$  on a state satisfying that predicate.

Today, we'll see how to compute weakest liberal preconditions (WLP) for IMP.

# VC generation with WP and SP

## $sp(S, P)$

- The strongest predicate that holds for states produced by executing  $S$  on a state satisfying  $P$ .

Symbolic execution, covered in next lecture, computes SPs for finite programs (no unbounded loops).

## $wp(S, Q)$

- The weakest predicate that guarantees  $Q$  will hold for states produced by executing  $S$  on a state satisfying that predicate.

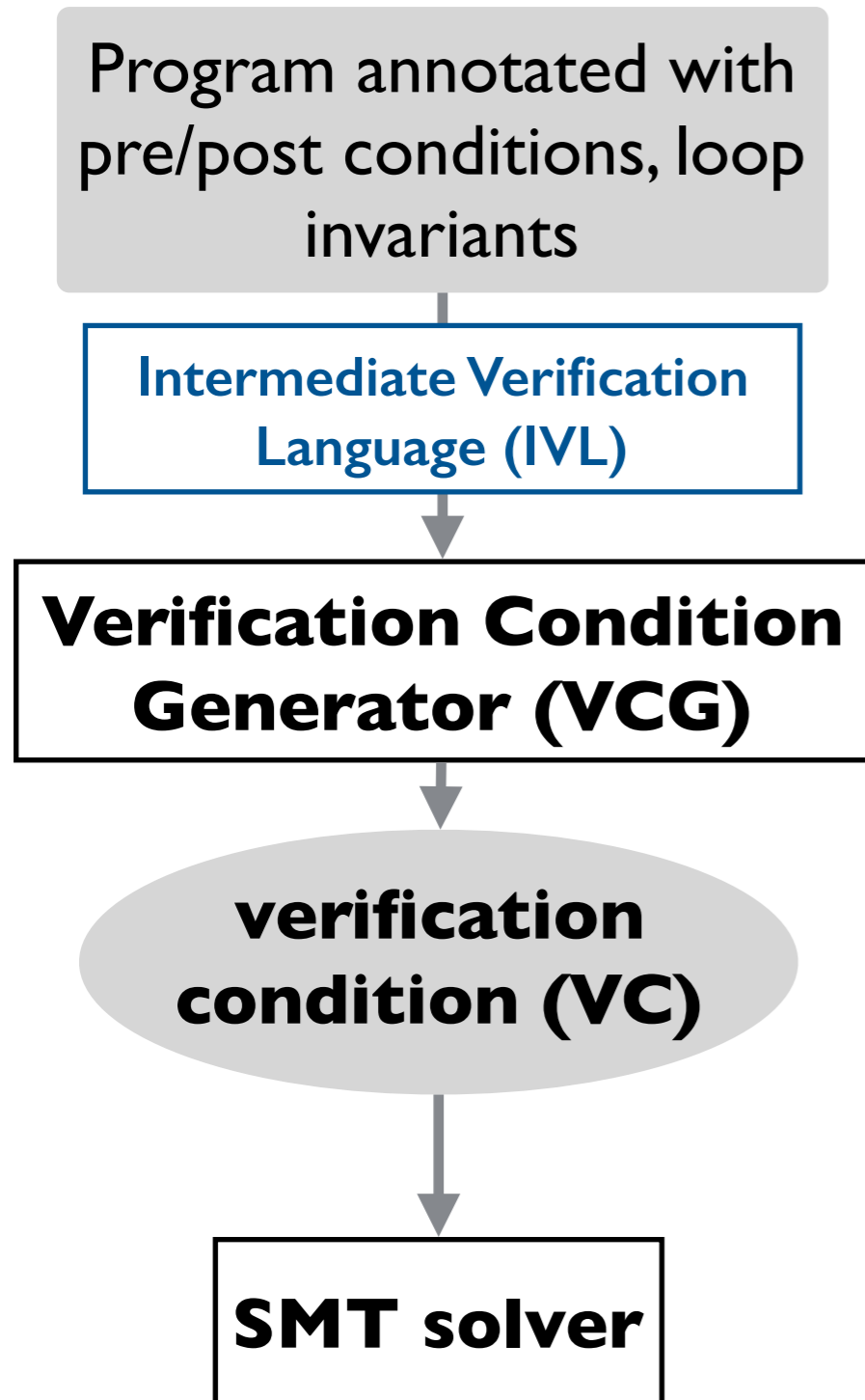
Today, we'll see how to compute weakest liberal preconditions (WLP) for IMP.

This lets us verify partial correctness properties.

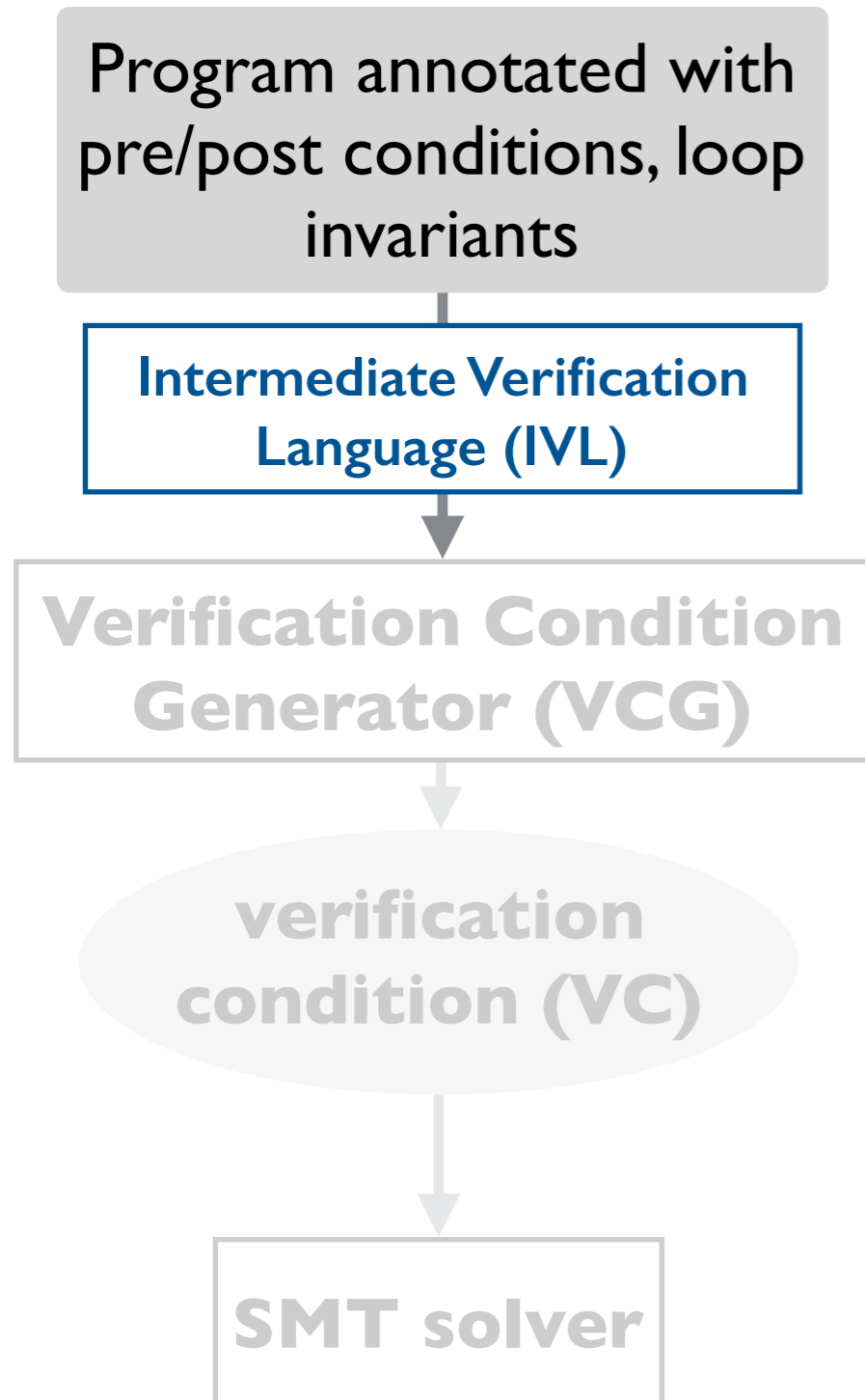
## $\{P\} S \{Q\}$ is valid if

- $P \Rightarrow wp(S, Q)$  or
- $sp(S, P) \Rightarrow Q$

# VC generation with WP



# VC generation with WP





# VC generation with WP: from IMP to IVL

Program annotated with pre/post conditions, loop invariants (IMP)

Intermediate Verification Language (IVL)

Verification Condition Generator (VCG)

verification condition (VC)

SMT solver

$E ::= Z \mid V \mid E + E \mid E * E$

$C ::= \text{true} \mid \text{false} \mid E = E \mid E \leq E$

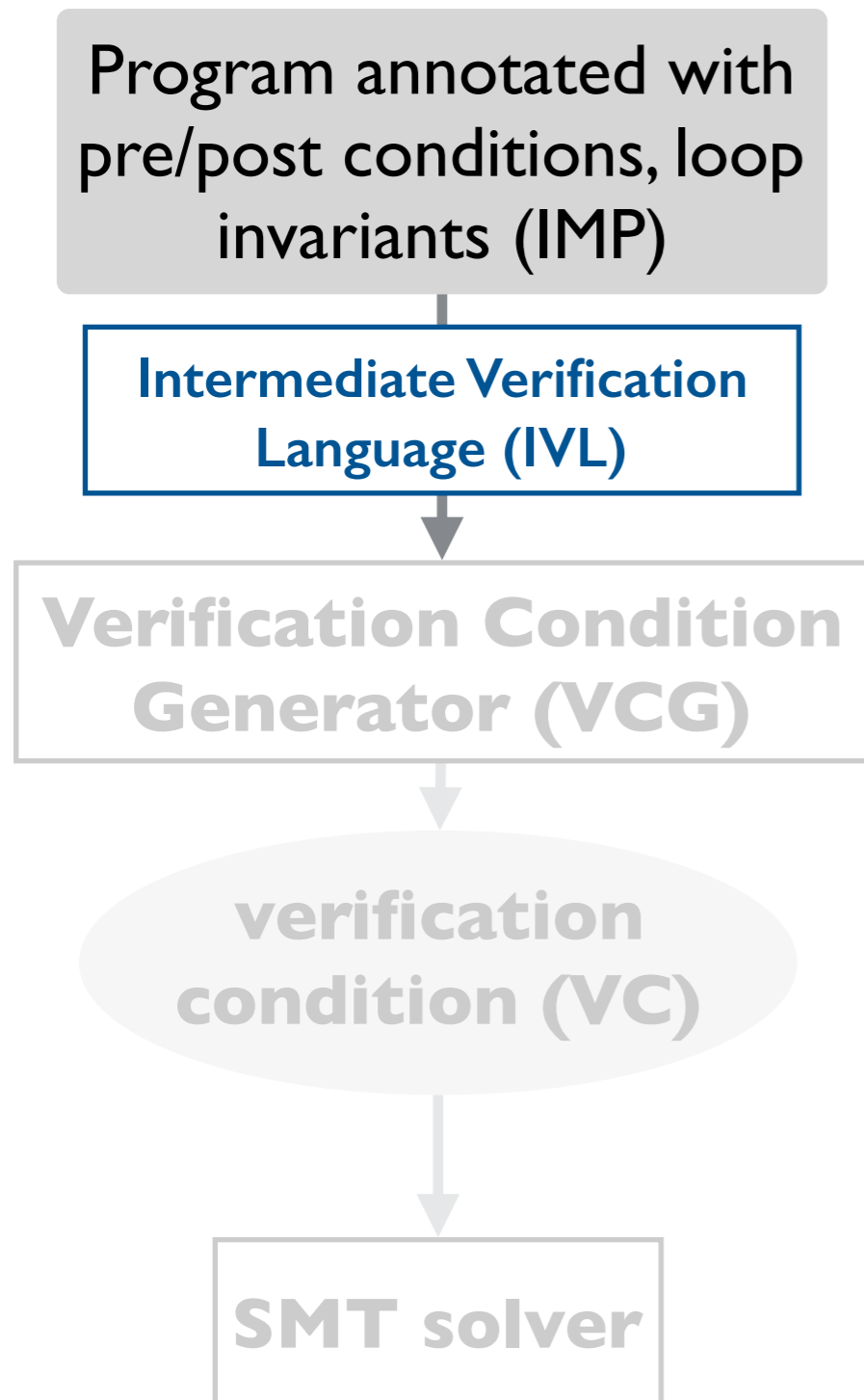
$S ::= \text{skip} \mid \text{abort} \mid V := E \mid S; S \mid$

**if C then S else S** |

**while C {I} do S**

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

# VC generation with WP: from IMP to IVL



$E ::= Z \mid V \mid E + E \mid E * E$

$C ::= \text{true} \mid \text{false} \mid E = E \mid E \leq E$

$S ::= \text{skip} \mid \text{abort} \mid V := E \mid S; S \mid$

**if C then S else S** |

**while C {I} do S**

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

$E ::= Z \mid V \mid E + E \mid E * E$

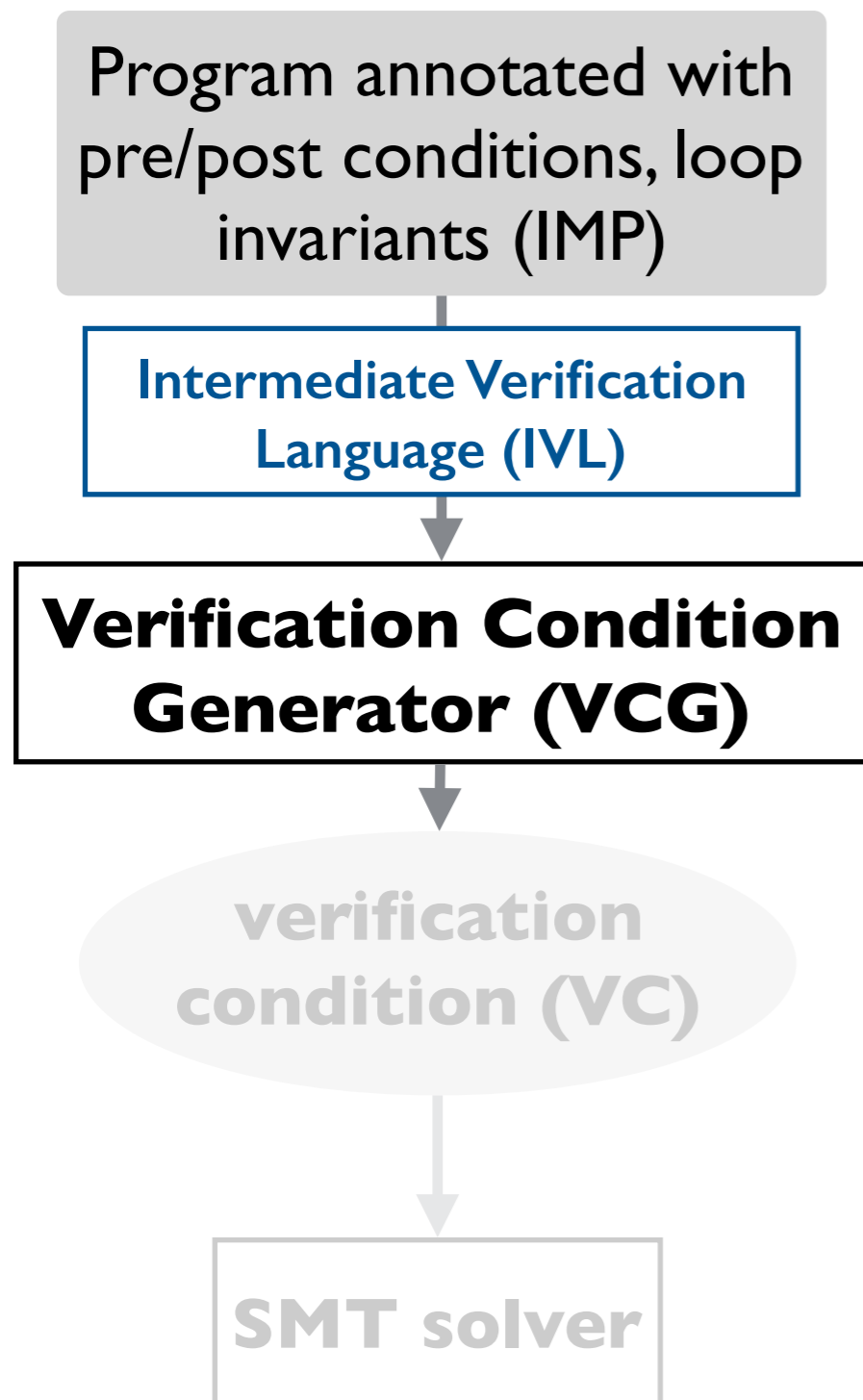
$C ::= \text{true} \mid \text{false} \mid E = E \mid E \leq E$

$S ::= \text{skip} \mid \text{abort} \mid V := E \mid S; S \mid$

**if C then S else S** |

**assert C** | **assume C** | **havoc V**

# VC generation with WP: loop-free code



$E ::= Z \mid V \mid E + E \mid E * E$  {P} S {Q}

$C ::= \text{true} \mid \text{false} \mid E = E \mid E \leq E$

$S ::= \text{skip} \mid \text{abort} \mid V := E \mid S; S \mid$

**if C then S else S** |

**while C {I} do S**



$E ::= Z \mid V \mid E + E \mid E * E$

$C ::= \text{true} \mid \text{false} \mid E = E \mid E \leq E$

$S ::= \text{skip} \mid \text{abort} \mid V := E \mid S; S \mid$

**if C then S else S** |

**assert C** | **assume C** | **havoc V**

# **VC generation with WP: loop-free code**

# **VC generation with WP: loop-free code**

**wp(S, Q):**

# VC generation with WP: loop-free code

**wp(S, Q):**

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

# VC generation with WP: loop-free code

**wp(S, Q):**

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

# VC generation with WP: loop-free code

**wp(S, Q):**

- $\text{wp}(\mathbf{skip}, Q) = Q$
- $\text{wp}(\mathbf{abort}, Q) = \text{true}$
- $\text{wp}(x := E, Q) = Q[E / x]$



# VC generation with WP: loop-free code

**wp(S, Q):**

- $\text{wp}(\mathbf{skip}, Q) = Q$
- $\text{wp}(\mathbf{abort}, Q) = \text{true}$
- $\text{wp}(x := E, Q) = Q[E / x]$
- $\text{wp}(S_1; S_2, Q) = \text{wp}(S_1, \text{wp}(S_2, Q))$

# VC generation with WP: loop-free code

## **wp(S, Q):**

- $\text{wp}(\mathbf{skip}, Q) = Q$
- $\text{wp}(\mathbf{abort}, Q) = \text{true}$
- $\text{wp}(x := E, Q) = Q[E / x]$
- $\text{wp}(S_1; S_2, Q) = \text{wp}(S_1, \text{wp}(S_2, Q))$
- $\text{wp}(\mathbf{if } C \mathbf{ then } S_1 \mathbf{ else } S_2, Q) = (C \rightarrow \text{wp}(S_1, Q)) \wedge (\neg C \rightarrow \text{wp}(S_2, Q))$

# VC generation with WP: what about loops?

## wp(S, Q):

- $\text{wp}(\mathbf{skip}, Q) = Q$
- $\text{wp}(\mathbf{abort}, Q) = \text{true}$
- $\text{wp}(x := E, Q) = Q[E / x]$
- $\text{wp}(S_1; S_2, Q) = \text{wp}(S_1, \text{wp}(S_2, Q))$
- $\text{wp}(\mathbf{if } C \mathbf{ then } S_1 \mathbf{ else } S_2, Q) = (C \rightarrow \text{wp}(S_1, Q)) \wedge (\neg C \rightarrow \text{wp}(S_2, Q))$
- $\text{wp}(\mathbf{while } C \mathbf{ \{ \} do } S, Q) = ?$

# VC generation with WP: what about loops?

## wp(S, Q):

- $\text{wp}(\mathbf{skip}, Q) = Q$
- $\text{wp}(\mathbf{abort}, Q) = \text{true}$
- $\text{wp}(x := E, Q) = Q[E / x]$
- $\text{wp}(S_1; S_2, Q) = \text{wp}(S_1, \text{wp}(S_2, Q))$
- $\text{wp}(\mathbf{if } C \mathbf{ then } S_1 \mathbf{ else } S_2, Q) = (C \rightarrow \text{wp}(S_1, Q)) \wedge (\neg C \rightarrow \text{wp}(S_2, Q))$
- $\text{wp}(\mathbf{while } C \mathbf{ \{ \} do } S, Q) = \mathbf{X}$

A fixpoint! In general, cannot be expressed as a syntactic construction in terms of the postcondition.

# VC generation with WP: what about loops?

## wp(S, Q):

- $\text{wp}(\mathbf{skip}, Q) = Q$
- $\text{wp}(\mathbf{abort}, Q) = \text{true}$
- $\text{wp}(x := E, Q) = Q[E / x]$
- $\text{wp}(S_1; S_2, Q) = \text{wp}(S_1, \text{wp}(S_2, Q))$
- $\text{wp}(\mathbf{if } C \mathbf{ then } S_1 \mathbf{ else } S_2, Q) = (C \rightarrow \text{wp}(S_1, Q)) \wedge (\neg C \rightarrow \text{wp}(S_2, Q))$
- $\text{wp}(\mathbf{while } C \{I\} \mathbf{ do } S, Q) = \mathbf{X}$

A fixpoint! In general, cannot be expressed as a syntactic construction in terms of the postcondition.

Use loop invariants to approximate loop behavior. Then check each invariant is correct and strong enough.

# VC generation with WP: what about loops?

**while** C {I} **do** S

Use loop invariants to approximate loop behavior. Then check each invariant is correct and strong enough.

# VC generation with WP: what about loops?

**while** C {I} **do** S



Cut the loop.

**assert** I;

**havoc** x; ... *// for each loop target x*

**assume** I;

**if** C **then** S; **assert** I; **assume** false;

**else skip;**

Use loop invariants to approximate loop behavior. Then check each invariant is correct and strong enough.

# VC generation with WP: what about loops?

**while** C {I} **do** S



Cut the loop.

**wp(S, Q):**

**assert** I;

**havoc** x; ... *// for each loop target x*

**assume** I;

**if** C **then** S; **assert** I; **assume** false;

**else skip;**

Use loop invariants to approximate loop behavior. Then check each invariant is correct and strong enough.



# VC generation with WP: what about loops?

**while** C {I} **do** S



Cut the loop.

**assert** I;

**havoc** x; ... *// for each loop target x*

**assume** I;

**if** C **then** S; **assert** I; **assume** false;

**else skip;**

**wp(S, Q):**

- $\text{wp}(\text{assert } C, Q) = C \wedge Q$

Use loop invariants to approximate loop behavior. Then check each invariant is correct and strong enough.

# VC generation with WP: what about loops?

**while** C {I} **do** S



Cut the loop.

**assert** I;

**havoc** x; ... *// for each loop target x*

**assume** I;

**if** C **then** S; **assert** I; **assume** false;

**else skip;**

**wp(S, Q):**

- $\text{wp}(\text{assert } C, Q) = C \wedge Q$
- $\text{wp}(\text{assume } C, Q) = C \rightarrow Q$

Use loop invariants to approximate loop behavior. Then check each invariant is correct and strong enough.

# VC generation with WP: what about loops?

**while** C {I} **do** S



Cut the loop.

**assert** I;

**havoc** x; ... *// for each loop target x*

**assume** I;

**if** C **then** S; **assert** I; **assume** false;  
**else skip**;

**wp(S, Q):**

- $\text{wp}(\text{assert } C, Q) = C \wedge Q$
- $\text{wp}(\text{assume } C, Q) = C \rightarrow Q$
- $\text{wp}(\text{havoc } x, Q) = \forall x. Q$

Use loop invariants to approximate loop behavior. Then check each invariant is correct and strong enough.

# VC generation with WP: putting it all together

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

- $\text{wp}(\mathbf{skip}, Q) = Q$
- $\text{wp}(\mathbf{abort}, Q) = \text{true}$
- $\text{wp}(x := E, Q) = Q[E / x]$
- $\text{wp}(S_1; S_2, Q) = \text{wp}(S_1, \text{wp}(S_2, Q))$
- $\text{wp}(\mathbf{if } C \mathbf{ then } S_1 \mathbf{ else } S_2, Q) = (C \rightarrow \text{wp}(S_1, Q)) \wedge (\neg C \rightarrow \text{wp}(S_2, Q))$
- $\text{wp}(\mathbf{assert } C, Q) = C \wedge Q$
- $\text{wp}(\mathbf{assume } C, Q) = C \rightarrow Q$
- $\text{wp}(\mathbf{havoc } x, Q) = \forall x. Q$

1. Translate IMP to IVL by cutting loops.
2. Compute WP for IVL.

# Verifying a Hoare triple

**Theorem:  $\{P\} S \{Q\}$  is valid if the following formula is valid**

$$P \rightarrow \text{wp}(S_{\text{IVL}}, Q)$$

# Verifying a Hoare triple

**Theorem:  $\{P\} S \{Q\}$  is valid if the following formula is valid**

$$P \rightarrow \text{wp}(S_{\text{IVL}}, Q)$$

The other direction doesn't hold because loop invariants may not be strong enough or they may be incorrect. Might get false alarms.

# Summary

## Today

- Automating Hoare Logic with VCG based on WPs

## Next lecture

- Symbolic execution