

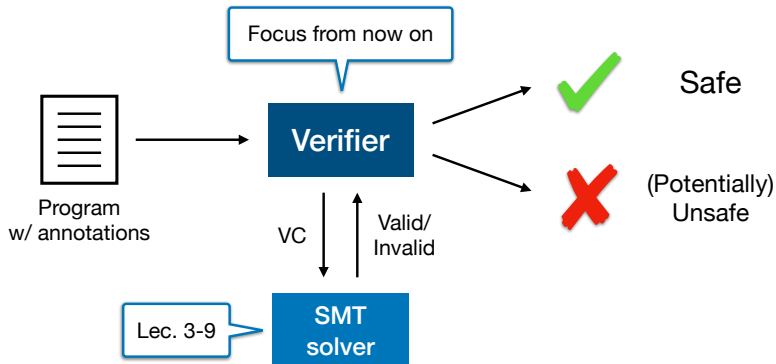
# EC4219: Software Engineering

## Lecture 10 — Program Verification (1) *Formal Specifications*

Sunbeom So  
2024 Spring

# Where We Are

- In Lec. 3–9, we have learned the basic internals of SMT solvers.
- In the following lectures, we will learn program verification by treating SMT solvers as black-boxes.



# Overview: Program Verification

We will learn methods for specifying and verifying program properties.

- **Specification (program annotation)**: precise statement of program properties in first-order logic (with some appropriate theory  $T$ ).

We focus on two forms of properties.

- ▶ **Partial correctness properties (safety properties)**

If the precondition of a function (or a program) is satisfied, its postcondition and assertions are satisfied if the function returns (halts).

- ▶ **Total correctness properties: partial correctness + termination**

If the input satisfies the function precondition, then the function eventually halts and produces the output satisfying the function postcondition.

- **Verification methods**: for proving partial/total correctness.

- ▶ **Inductive assertion method** for proving partial correctness
- ▶ **Ranking function method** for proving total correctness

We will focus on proving **partial correctness**.

# Specification

- An annotation is a first-order logic formula  $F$  (in some appropriate theory  $T$ ).
- An annotation  $F$  at location  $L$  expresses an **invariant** asserting that  $F$  is *true* whenever program control reaches  $L$ .
- Three major kinds of annotations.
  - 1 Function specification
  - 2 Loop invariant
  - 3 Assertion

# Function Specifications

Formulas whose free variables include only the formal parameters and return variables.

- **Precondition:** Specification about what should be true upon entering the function.
- **Postcondition:** Specification about the expected output of the function. The postcondition relates the function's output (the return value) to its input (the formal parameters).

# Function Specification Example 1: Linear Search

```
1  bool LinearSearch (int  $a[]$ , int  $l$ , int  $u$ , int  $e$ ) {  
2      int  $i := l$ ;  
3      while ( $i \leq u$ ) {  
4          if ( $a[i] = e$ ) return true;  
5           $i := i + 1$ ;  
6      }  
7      return false;  
8  }
```

Q. Does this function always behave correctly?

# Function Specification Example 1: Linear Search

```
1  @pre:  $0 \leq l \wedge u < |a|$   
2  @post:  $rv \leftrightarrow \exists i. l \leq i \leq u \wedge a[i] = e$   
3  bool LinearSearch (int  $a[]$ , int  $l$ , int  $u$ , int  $e$ ) {  
4      int  $i := l$ ;  
5      while ( $i \leq u$ ) {  
6          if ( $a[i] = e$ ) return true;  
7           $i := i + 1$ ;  
8      }  
9      return false;  
10 }
```

- It behaves correctly only when  $l \geq 0$  and  $u < |a|$ .
- It returns *true* iff the array  $a$  contains  $e$  in the range  $[l, u]$ .

Our goal is to prove the **partial correctness** property; if the function precondition holds and the function halts, then its postcondition holds upon return.

## Function Specification Example 2: Binary Search

```
1  @pre:
2  @post:
3  bool BinarySearch (int  $a[]$ , int  $l$ , int  $u$ , int  $e$ ) {
4      if ( $l > u$ ) return false;
5      else {
6          int  $m := (l + u) \text{ div } 2$ ;
7          if ( $a[m] = e$ ) return true;
8          else if ( $a[m] < e$ ) return BinarySearch ( $a, m + 1, u, e$ )
9          else return BinarySearch ( $a, l, m - 1, e$ )
10     }
11 }
```

- It behaves correctly only when
- It returns *true* iff



## Function Specification Example 2: Binary Search

```
1  @pre:  $0 \leq l \wedge u < |a| \wedge \text{sorted}(a, l, u)$ 
2  @post:  $rv \leftrightarrow \exists i. l \leq i \leq u \wedge a[i] = e$ 
3  bool BinarySearch (int  $a[]$ , int  $l$ , int  $u$ , int  $e$ ) {
4      if ( $l > u$ ) return false;
5      else {
6          int  $m := (l + u) \text{ div } 2$ ;
7          if ( $a[m] = e$ ) return true;
8          else if ( $a[m] < e$ ) return BinarySearch ( $a, m + 1, u, e$ )
9          else return BinarySearch ( $a, l, m - 1, e$ )
10     }
11 }
```

- It behaves correctly only when  $l \geq 0$ ,  $u < |a|$ , and  $a$  is sorted.
- It returns *true* iff the array  $a$  contains  $e$  in the range  $[l, u]$ .

The predicate **sorted** is defined in the combined theory of integers and arrays ( $T_{\mathbb{Z}} \cup T_A$ ).

$$\text{sorted}(a, l, u) \iff \forall i, j. l \leq i \leq j \leq u \rightarrow a[i] \leq a[j]$$

# Function Specification Example 3: Bubble Sort

```
1  @pre:
2  @post:
3  bool BubbleSort (int a0[]) {
4      int[] a := a0;
5      for (int i := |a| - 1; i > 0; i := i - 1) {
6          for (int j := 0; j < i; j := j + 1) {
7              if (a[j] > a[j + 1]) {
8                  int t := a[j];
9                  a[j] := a[j + 1];
10                 a[j + 1] := t;
11             }
12         }
13     }
14     return a;
15 }
```

BubbleSort works by “bubbling” the largest element of the left unsorted region of the array, toward the sorted region on the right.

- Any array can be given as input.
- The returned array is sorted.

## Function Specification Example 3: Bubble Sort

```
1  @pre:  $\top$ 
2  @post:  $\text{sorted}(rv, 0, |rv| - 1)$ 
3  bool BubbleSort (int  $a_0[]$ ) {
4      int[]  $a := a_0$ ;
5      for (int  $i := |a| - 1$ ;  $i > 0$ ;  $i := i - 1$ ) {
6          for (int  $j := 0$ ;  $j < i$ ;  $j := j + 1$ ) {
7              if ( $a[j] > a[j + 1]$ ) {
8                  int  $t := a[j]$ ;
9                   $a[j] := a[j + 1]$ ;
10                  $a[j + 1] := t$ ;
11             }
12         }
13     }
14     return  $a$ ;
15 }
```

BubbleSort works by “bubbling” the largest element of the left unsorted region of the array, toward the sorted region on the right.

- Any array can be given as input.
- The returned array is sorted.

# Necessity of Loop Summarization

```
1  @pre:  $n \geq 0$   
2  @post:  $j = n$   
3  bool Loop (int  $n$ ) {  
4      int  $i := 0$ ;  
5      int  $j := 0$ ;  
6      while ( $i \neq n$ ) {  
7           $i := i + 1$ ;  
8           $j := j + 1$ ;  
9      }  
10     return;  
11 }
```

Q1. Does this function satisfy the function specification?

# Necessity of Loop Summarization

```
1  @pre:  $n \geq 0$   
2  @post:  $j = n$   
3  bool Loop (int  $n$ ) {  
4      int  $i := 0$ ;  
5      int  $j := 0$ ;  
6      while ( $i \neq n$ ) {  
7           $i := i + 1$ ;  
8           $j := j + 1$ ;  
9      }  
10     return;  
11 }
```

Q1. Does this function satisfy the function specification?

Q2. How can you formally ensure that?

# Necessity of Loop Summarization

```
1  @pre:  $n \geq 0$ 
2  @post:  $j = n$ 
3  bool Loop (int  $n$ ) {
4      int  $i := 0$ ;
5      int  $j := 0$ ;
6      while ( $i \neq n$ ) {
7           $i := i + 1$ ;
8           $j := j + 1$ ;
9      }
10     return;
11 }
```

Q1. Does this function satisfy the function specification?

Q2. How can you formally ensure that?

We need to summarize the behaviors of loops. In our example,  $i = j$  is the summarization that is precise enough to prove the correctness.

$$i = j \wedge i = n \rightarrow j = n$$

# Loop Invariant

To prove partial correctness, each loop often needs to be annotated with a proper loop invariant  $F$ .

```
1 while
2   @ $F$ 
3   ( $\langle condition \rangle$ ) {
4    $\langle body \rangle$ 
5 }
```

Loop invariant  $F$  is a property that holds before the entrance and is preserved by executions of the loop body. In other words,  $F$  holds at the beginning of every iteration. Therefore,

- $F \wedge \langle condition \rangle$  holds when entering the body.
- $F \wedge \neg \langle condition \rangle$  holds when exiting the loop.

# Loop Invariant Example 1: Linear Search

Find a nontrivial<sup>1</sup> loop invariant of the loop in LinearSearch.

```
1  pre:  $0 \leq l \wedge u < |a|$ 
2  @post:  $rv \leftrightarrow \exists i. l \leq i \leq u \wedge a[i] = e$ 
3  bool LinearSearch (int  $a[]$ , int  $l$ , int  $u$ , int  $e$ ) {
4      int  $i := l$ ;
5      while
6      @L:  $l \leq i \wedge (\forall j. l \leq j < i \rightarrow a[j] \neq e)$ 
7      ( $i \leq u$ ) {
8          if ( $a[i] = e$ ) return true;
9           $i := i + 1$ ;
10     }
11     return false;
12 }
```

- The index  $i$  is at least  $j$ .
- We have not find an element with the previously examined indices  $j$ .

---

<sup>1</sup>A trivial loop invariant is *true*, which is useless in most cases.



# Loop Invariant Example 2: Bubble Sort

```
1  @pre:  $\top$ 
2  @post:  $\text{sorted}(rv, 0, |rv| - 1)$ 
3  bool BubbleSort (int  $a[]$ ) {
4    int[]  $a := a_0$ ;
5    @ $L_1$  :  $\left\{ \begin{array}{l} -1 \leq i < |a| \\ \wedge \text{partitioned}(a, 0, i, i + 1, |a| - 1) \\ \wedge \text{sorted}(a, i, |a| - 1) \end{array} \right\}$ 
6    for (int  $i := |a| - 1$ ;  $i > 0$ ;  $i := i - 1$ ) {
7      @ $L_2$  :  $\left\{ \begin{array}{l} 1 \leq i < |a| \wedge 0 \leq j \leq i \\ \wedge \text{partitioned}(a, 0, i, i + 1, |a| - 1) \\ \wedge \text{partitioned}(a, 0, j - 1, j, j) \\ \wedge \text{sorted}(a, i, |a| - 1) \end{array} \right\}$ 
8      for (int  $j := 0$ ;  $j < i$ ;  $j := j + 1$ ) {
9        if ( $a[j] > a[j + 1]$ ) {
10          int  $t := a[j]$ ;
11           $a[j] := a[j + 1]$ ;
12           $a[j + 1] := t$ ;
13        }
14      }
15    }
16    return  $a$ ;
17 }
```

$\text{partitioned}(a, l_1, u_1, l_2, u_2) \iff \forall i, j. l_1 \leq i \leq u_1 < l_2 \leq j \leq u_2 \rightarrow a[i] \leq a[j]$

# Assertions

- The other formal comments on expected program behaviors.
- Usually specified by the command `assert` in most programming languages. If assertion violations occur at runtime, typically raise exceptions; very useful for debugging errors.

```
1  @pre:  $0 \leq l \wedge u < |a|$ 
2  @post:  $rv \leftrightarrow \exists i. l \leq i \leq u \wedge a[i] = e$ 
3  bool LinearSearch (int  $a[]$ , int  $l$ , int  $u$ , int  $e$ ) {
4      int  $i := l$ ;
5      while
6          @L :  $l \leq i \wedge (\forall j. l \leq j < i \rightarrow a[j] \neq e)$ 
7          ( $i \leq u$ ) {
8          @0  $0 \leq i < |a|$  /* expectation: array access is legal */
9              if ( $a[i] = e$ ) return true;
10              $i := i + 1$ ;
11         }
12     return false;
13 }
```

## cf) Runtime Assertions

- A special class of assertions automatically inserted by compilers to catch runtime errors.
  - ▶ division-by-zero, null-dereference, accessing an array out of bounds, etc.
- For example, given the C command below

$\dots; i := i/j; \dots$

we should interpret it as

$\dots; i := i/j; \text{assert}(j \neq 0); \dots$

# Summary

- Goal: prove the “correctness” of implementations
- We learned specification methods to rigorously describe the “correct” behaviors.
  - ① Function specification: precondition, postcondition
  - ② Loop invariant: summarization of loops
  - ③ Assertion: The other formal comments on expected behaviors

Q. how can we prove that our implementations obey the specifications?  
How to prove the **partial correctness**?

A. **Inductive assertion method** (Next class!)