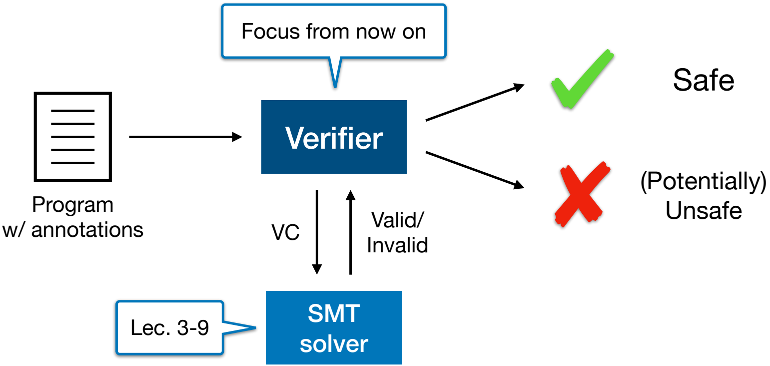
EC4219: Software Engineering

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

```
bool LinearSearch (int a[], int l, int u, int e) {
  int i := l;
  while (i ≤ u) {
    if (a[i] = e) return true;
    i := i + 1;
  }
  return false;
}
```

Q. Does this function always behave correctly?

Function Specification Example 1: Linear Search

```
@pre: 0 ≤ l ∧ u < |a|
@post: rv ↔ ∃i.l ≤ i ≤ u ∧ a[i] = e
bool LinearSearch (int a[], int l, int u, int e) {
  int i := l;
  while (i ≤ u) {
    if (a[i] = e) return true;
    i := i + 1;
  }
  return false;
}
```

- 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

```
@pre:
@post:
bool BinarySearch (int a[], int l, int u, int e) {
  if (l > u) return false;
  else {
    int m := (l + u) div 2;
    if (a[m] = e) return true;
    else if (a[m] < e) return BinarySearch (a, m + 1, u, e)
    else return BinarySearch (a, l, m - 1, e)
  }
}
```

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

Function Specification Example 2: Binary Search

```
@pre: 0 ≤ l ∧ u < |a| ∧ sorted(a, l, u)
@post: rv ↔ ∃i.l ≤ i ≤ u ∧ a[i] = e
bool BinarySearch (int a[], int l, int u, int e) {
  if (l > u) return false;
  else {
    int m := (l + u) div 2;
    if (a[m] = e) return true;
    else if (a[m] < e) return BinarySearch (a, m + 1, u, e)
    else return BinarySearch (a, l, m − 1, e)
  }
}
```

- 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 ($T_{\mathbb{Z}} \cup T_A$).

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

Function Specification Example 3: Bubble Sort

```
@pre:
@post:
bool BubbleSort (int a0[]) {
  int[] a := a0;
  for (int i := |a| - 1; i > 0; i := i - 1) {
    for (int j := 0; j < i; j := j + 1) {
      if (a[j] > a[j + 1]) {
        int t := a[j];
        a[j] := a[j + 1];
        a[j + 1] := t;
      }
    }
  }
  return a;
}
```

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

```
@pre: ⊤
@post: sorted(rv, 0, |rv| − 1)
bool BubbleSort (int a0[]) {
  int[] a := a0;
  for (int i := |a| − 1; i > 0; i := i − 1) {
    for (int j := 0; j < i; j := j + 1) {
      if (a[j] > a[j + 1]) {
        int t := a[j];
        a[j] := a[j + 1];
        a[j + 1] := t;
      }
    }
  }
  return a;
}
```

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

```
@pre: ⊤
@post: j = n
bool Loop (int n) {
  int i := 0;
  int j := 0;
  while (i < n) {
    i := i + 1;
    j := j + 1;
  }
  return;
}
```

Q1. Does this function satisfy the function specification?

```
@pre: ⊤
@post: j = n
bool Loop (int n) {
  int i := 0;
  int j := 0;
  while (i < n) {
    i := i + 1;
    j := j + 1;
  }
  return;
}
```

Q1. Does this function satisfy the function specification?

Q2. How can you formally ensure that?

Necessity of Loop Summarization

```
@pre: ⊤
@post: j = n
bool Loop (int n) {
  int i := 0;
  int j := 0;
  while (i < n) {
    i := i + 1;
    j := j + 1;
  }
  return;
}
```

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.

$$i = j \land i \geq n \rightarrow j = n$$

Loop Invariant

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

```
while
  @F
  (⟨condition⟩) {
  ⟨body⟩
}
```

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,

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

Loop Invariant Example 1: Linear Search

Find a nontrivial[1] loop invariant of the loop in LinearSearch.

```
@pre: 0 ≤ l ∧ u < |a|
@post: rv ↔ ∃i.l ≤ i ≤ u ∧ a[i] = e
bool LinearSearch (int a[], int l, int u, int e) {
  int i := l;
  while
  @L : l ≤ i ∧ (∀j.l ≤ j < i → a[j] ≠ e)
  (i ≤ u) {
    if (a[i] = e) return true;
    i := i + 1;
  }
  return false;
}
```

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

[1] A trivial loop invariant is $true$, which is useless in most cases.

Loop Invariant Example 2: Bubble Sort

@pre: $\top$

@post: $\textbf{sorted}(rv, 0, |rv| - 1)$

```
bool BubbleSort (int a[]) {
  int[] a := a0;
```

$$@L_1 : \left\{ \begin{array}{l} -1 \le i < |a| \\ \wedge\ \textbf{partitioned}(a, 0, i, i+1, |a|-1) \\ \wedge\ \textbf{sorted}(a, i, |a|-1) \end{array} \right\}$$

```
  for (int i := |a| - 1; i > 0; i := i - 1) {
```

$$@L_2 : \left\{ \begin{array}{l} 1 \le i < |a| \wedge 0 \le j \le i \\ \wedge\ \textbf{partitioned}(a, 0, i, i+1, |a|-1) \\ \wedge\ \textbf{partitioned}(a, 0, j-1, j, j) \\ \wedge\ \textbf{sorted}(a, i, |a|-1) \end{array} \right\}$$

```
    for (int j := 0; j < i; j := j + 1) {
      if (a[j] > a[j + 1]) {
        int t := a[j];
        a[j] := a[j + 1];
        a[j + 1] := t;
      }
    }
  }
  return a;
}
```

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

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.

```
@pre: 0 ≤ l ∧ u < |a|
@post: rv ↔ ∃i.l ≤ i ≤ u ∧ a[i] = e
bool LinearSearch (int a[], int l, int u, int e) {
  int i := l;
  while
  @L : l ≤ i ∧ (∀j.l ≤ j < i → a[j] ≠ e)
  (i ≤ u) {
  @0 ≤ i < |a| /* expectation: array access is legal */
    if (a[i] = e) return true;
    i := i + 1;
  }
  return false;
}
```

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

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

we should interpret it as

$$\ldots;\ i := i/j;\ \texttt{assert}(j \neq 0);\ \ldots$$

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!)