SWE3002-42: Introduction to Software Engineering

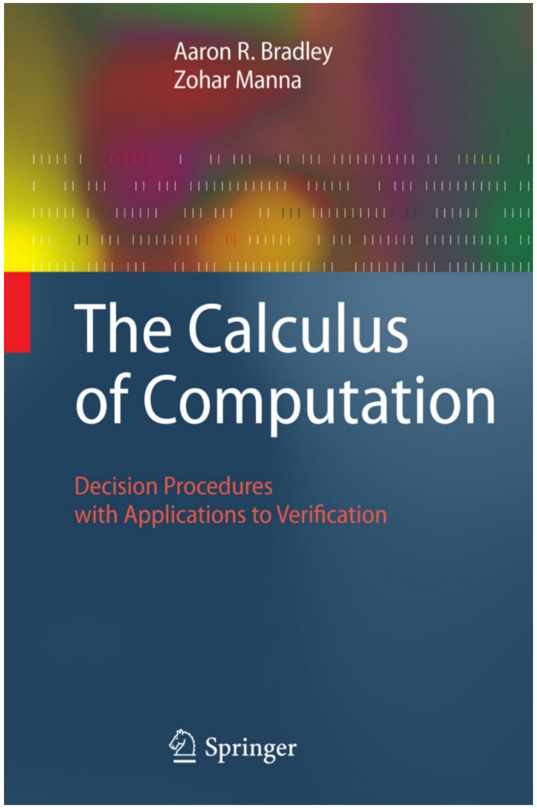
Lecture 12 – Program Verification

Sooyoung Cha

Department of Computer Science and Engineering

Program Verification

- Background
  - First-order logic
- Specification
  - pre-/post-conditions
  - Loop Invariant
  - Assertion
- Partial Correctness
  - Basic Paths
  - Weakest Precondition
  - Verification Conditions



Program Verification

- Let's prove that the program works as intended.

```
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
}
```

"Searching the range $[l, \mathrm{u}]$ of an array a of integers for a value e."

- ex) LinearSearch([1,3,5], 0, 2, 5) → true
- ex2) LinearSearch([1,3,5], 0, 2, 2) → false

Program Verification

- Techniques for specifying and verifying program properties.
  - **Specification (program annotations)**: precise statement of properties in first-order logic
    - Partial correctness properties (if a program halts, then its output satisfies some relation with its input.)
    - Total correctness properties
  - **Verification methods**:

Program Verification

- Techniques for specifying and verifying program properties.
  - **Specification (program annotations)**: precise statement of properties in first-order logic
    - Partial correctness properties (if a program halts, then its output satisfies some relation with its input.)
    - Total correctness properties
  - **Verification methods**: for proving partial/total correctness
    - Inductive assertion method
    - Ranking function method

```
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
}
```

Specification (Program Annotation)

- An annotation = A first-order logic formula *F*.
- First-order logic (FOL)
  - FOL is expressive enough to reason about programs.
  - Syntax

$\neg F$	negation ("not")	ex) $(F = \text{True}) \leftrightarrow (\neg F = \text{False})$
$F_1 \wedge F_2$	conjunction ("and")	ex) $(\text{True} \wedge \text{False}) \leftrightarrow \text{False}$
$F_1 \vee F_2$	disjunction ("or")	ex) $(\text{True} \vee \text{False}) \leftrightarrow \text{True}$
$F_1 \rightarrow F_2$	implication ("implies")	ex) if $(F_1 = \text{False})$ or $(F_2 = \text{True})$, then $(F_1 \rightarrow F_2)$ is True

Specification (Program Annotation)

- An annotation = A first-order logic formula *F*.
- First-order logic (FOL)
  - FOL is expressive enough to reason about programs.
  - Syntax

$\neg F$	negation ("not")	
$F_1 \land F_2$	conjunction ("and")	
$F_1 \lor F_2$	disjunction ("or")	
$F_1 \rightarrow F_2$	implication ("implies")	
$F_1 \leftrightarrow F_2$	iff ("if and only if")	ex) if ($F_1 = F_2 =$ False) or ($F_1 = F_2 =$ True)
$\exists x.F[x]$	existential quantification	ex) $\exists x.(x*x = 4) \leftrightarrow$ True
$\forall x.F[x]$	universal quantification	ex) $\forall x.(x*x = 4) \leftrightarrow$ False

Specification (Program Annotation)

- Three types of annotations:
  - Function specification
  - Loop invariant
  - Assertion

```
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
}
```

Formal parameters: array a, integer l, int u, int e

Function Specifications

- A pair of annotation
  - Precondition:
    - Specification about what should be true upon entering the function. (using the formal parameters)
  - Postcondition:
    - Specification about the expected output of the function. (using the formal parameters and the return variables of the function.)

```
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
}
```

Example 1: Linear Search

- Precondition and postcondition of LinearSearch?

$$
\begin{array}{l}
\texttt{bool}\ \mathsf{LinearSearch}(\texttt{int}[]\ a,\ \texttt{int}\ \ell,\ \texttt{int}\ u,\ \texttt{int}\ e)\ \{\\
\quad \texttt{for}\ @\ \top\\
\qquad (\texttt{int}\ i := \ell;\ i \leq u;\ i := i+1)\ \{\\
\qquad \texttt{if}\ (a[i] = e)\ \texttt{return true};\\
\quad \}\\
\quad \texttt{return false};\\
\}
\end{array}
$$

Example 1: Linear Search

- Precondition and postcondition of LinearSearch?
  - It behaves correctly only when $0 \leq l$ and $u < |a|$.
  - It returns true iff the array a contains the value e in the range $[l, u]$.

$$
\begin{array}{l}
@\text{pre } 0 \leq \ell \ \wedge \ u < |a| \\
@\text{post } rv \ \leftrightarrow \ \exists i.\ \ell \leq i \leq u \ \wedge \ a[i] = e \\
\texttt{bool}\ \text{LinearSearch}(\texttt{int}[]\ a,\ \texttt{int}\ \ell,\ \texttt{int}\ u,\ \texttt{int}\ e)\ \{ \\
\quad \texttt{for}\ @\ \top \\
\qquad (\texttt{int}\ i := \ell;\ i \leq u;\ i := i + 1)\ \{ \\
\qquad \texttt{if}\ (a[i] = e)\ \texttt{return true}; \\
\quad \} \\
\quad \texttt{return false}; \\
\}
\end{array}
$$

Loop Invariant

- For proving partial correctness, each loop must be annotated with a loop invariant F:
  - Loop: applying the $<body>$ as long as $<condition>$ holds.

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

- Loop invariant F must hold at the beginning of every iteration:
  - $F \wedge <condition>$ holds on entering the body.
  - $F \wedge \neg <condition>$ holds when exiting the loop.

Loop Invariant

- Find a loop invariant of the loop in LinearSearch:

$@\text{pre} : 0 \leq l \land u < |a|$

$@\text{post} : rv \leftrightarrow \exists i. l \leq i \leq u \land a[i] = e$

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

Loop Invariant

- Find a loop invariant of the loop in LinearSearch:

$$
\begin{array}{l}
\textsf{@pre} : 0 \leq l \wedge u < |a| \\
\textsf{@post} : rv \leftrightarrow \exists i. l \leq i \leq u \wedge a[i] = e \\
\textsf{bool LinearSearch (int } a[], \textsf{ int } l, \textsf{ int } u, \textsf{ int } e) \ \{ \\
\quad \textsf{int } i := l; \\
\quad \textsf{while} \\
\quad @L : l \leq i \wedge (\forall j.\ l \leq j < i \rightarrow a[j] \neq e) \\
\quad (i \leq u) \ \{ \\
\quad\quad \textsf{if } (a[i] = e) \textsf{ return true} \\
\quad\quad i := i + 1; \\
\quad \} \\
\quad \textsf{return false} \\
\}
\end{array}
$$

Assertions

- Programmers' formal comments on the program behavior.
- Runtime assertions: division by 0, array out of bounds, etc

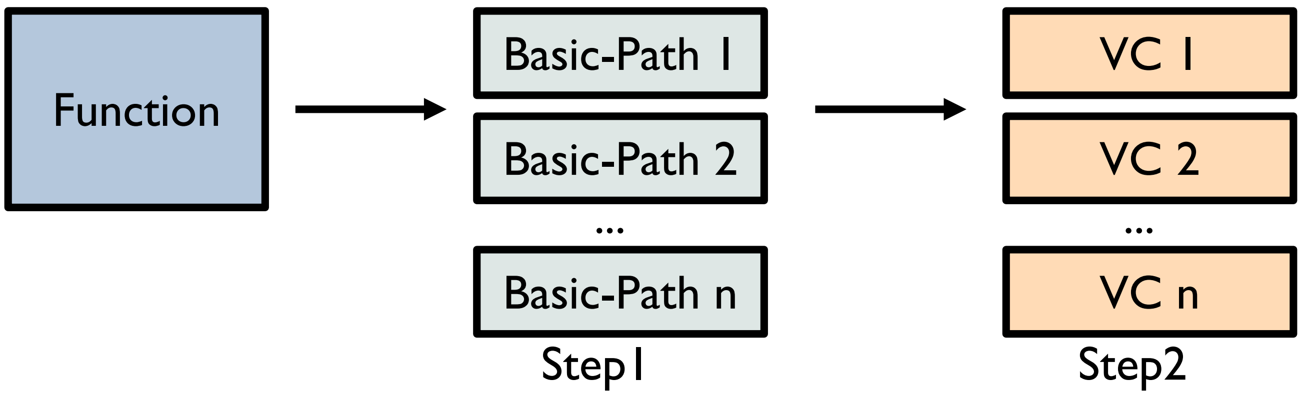
```
@pre : 0 ≤ l ∧ u < |a|
@post : rv ↔ ∃i.l ≤ i ≤ u ∧ a[i] = e
bool LinearSearch (int a[], int l, int u, int e) {
  int i := l;
  while
  @L : l ≤ i ∧ (∀j. l ≤ j < i → a[j] ≠ e)
  (i ≤ u) {
  @0 ≤ i < |a|
    if (a[i] = e) return true
    i := i + 1;
  }
  return false
}
```

Proving Partial Correctness

- Motivation
  - Does our program (e.g., function) work as we intend?
- Partial correctness
  - A function is *partially correct* if when the function's precondition is satisfied on entry, its postcondition is satisfied when the function returns.
- Inductive assertion method
  - Derive verification conditions (VCs) from a function.
  - Check the validity of VCs by an SMT solver.
  - If all of VCs are valid, the function is partially correct.

Deriving Verification Conditions (VCs)

- Done in two steps:
  - The function is broken down into a finite set of basic paths.
  - Each basic path generates a verification condition.



- Difficulty: Loops and recursive functions complicate proofs as they create an unbounded number of paths.
  - For loops, loop invariants cut the paths into a finite set of basic paths.
  - For recursion, function specification cuts the paths.

Basic Paths

- A sequence of atomic statements that begins at the function precondition or a loop invariant and ends at a loop invariant or the function postcondition.

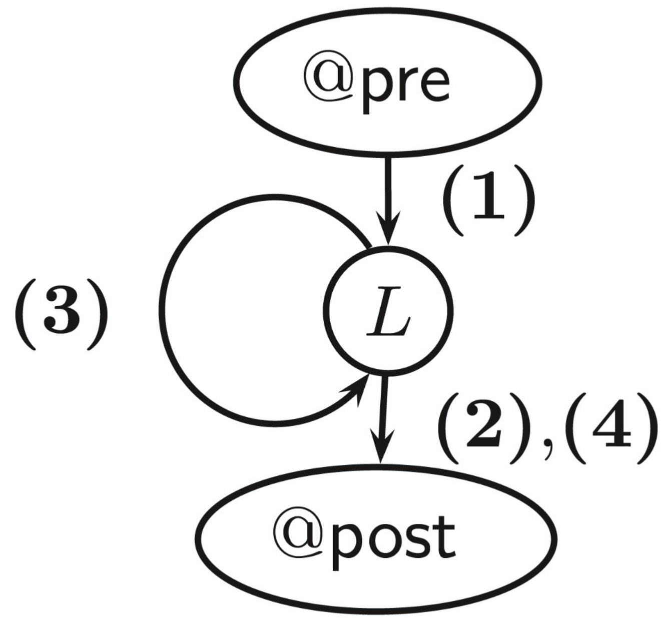
```
@pre : 0 ≤ l ∧ u < |a|
@post : rv ↔ ∃i.l ≤ i ≤ u ∧ a[i] = e
bool LinearSearch (int a[], int l, int u, int e) {
  int i := l;
  while
  @L : l ≤ i ∧ (∀j. l ≤ j < i → a[j] ≠ e)
  (i ≤ u) {
  @0 ≤ i < |a|
    if (a[i] = e) return true
    i := i + 1;
  }
  return false
}
```

Basic Paths

- A sequence of atomic statements that begins at the function precondition or a loop invariant and ends at a loop invariant or the function postcondition.

$@\mathsf{pre} : 0 \leq l \wedge u < |a|$
$@\mathsf{post} : rv \leftrightarrow \exists i. l \leq i \leq u \wedge a[i] = e$

bool LinearSearch (int $a[]$, int l, int u, int e) {
  int $i := l$;
  while
  $@L : l \leq i \wedge (\forall j.\, l \leq j < i \rightarrow a[j] \neq e)$
  $(i \leq u)$ {
  $@0 \leq i < |a|$
    if $(a[i] = e)$ return true
    $i := i + 1$;
  }
  return false
}



Basic Paths

- A basic path is an sequence of atomic statements that begins at the function precondition or a loop invariant and ends at a loop invariant or the function postcondition.

$@\text{pre} : 0 \leq l \wedge u < |a|$
$@\text{post} : rv \leftrightarrow \exists i. l \leq i \leq u \wedge a[i] = e$
bool LinearSearch (int $a[]$, int l, int u, int e) {
  int $i := l$;
  while
  $@L : l \leq i \wedge (\forall j.\ l \leq j < i \rightarrow a[j] \neq e)$
  $(i \leq u)$ {
  $@0 \leq i < |a|$
    if $(a[i] = e)$ return true
    $i := i + 1$;
  }
  return false
}

(1)
$@\text{pre} : 0 \leq l \wedge u < |a|$
$i := l$;
$@L : l \leq i \wedge (\forall j.\ l \leq j < i \rightarrow a[j] \neq e)$

(2)
$@L : l \leq i \wedge (\forall j.\ l \leq j < i \rightarrow a[j] \neq e)$
assume $i \leq u$;
assume $a[i] = e$;
$rv :=$ true
$@\text{post} : rv \leftrightarrow \exists i. l \leq i \leq u \wedge a[i] = e$

Basic Paths

- A sequence of atomic statements that begins at the function precondition or a loop invariant and ends at a loop invariant or the function postcondition.

$@\text{pre} : 0 \leq l \wedge u < |a|$
$@\text{post} : rv \leftrightarrow \exists i. l \leq i \leq u \wedge a[i] = e$
bool LinearSearch (int $a[]$, int l, int u, int e) {
  int $i := l$;
  while
  $@L : l \leq i \wedge (\forall j.\ l \leq j < i \rightarrow a[j] \neq e)$
  $(i \leq u)$ {
  $@0 \leq i < |a|$
    if $(a[i] = e)$ return true
    $i := i + 1$;
  }
  return false
}

(3)
$@L : l \leq i \wedge (\forall j.\ l \leq j < i \rightarrow a[j] \neq e)$
assume $i \leq u$;
assume $a[i] \neq e$
$i := i + 1$;
$@L : l \leq i \wedge (\forall j.\ l \leq j < i \rightarrow a[j] \neq e)$

(4)
$@L : l \leq i \wedge (\forall j.\ l \leq j < i \rightarrow a[j] \neq e)$
assume $i > u$;
$rv :=$ false
$@\text{post} : rv \leftrightarrow \exists i. l \leq i \leq u \wedge a[i] = e$

Verification Conditions

- Weakest Precondition
  - What is the precondition that must hold before the statement to ensure that the postcondition holds afterwards?
    - { ? } x:=x+1 { x > 0 }
    - { ? } y:=2*y { y < 5 }
    - { ? } x:=x+y { y > x }

Verification Conditions

- Weakest Precondition
  - What is the precondition that must hold before the statement to ensure that the postcondition holds afterwards?
    - $\{\ x > -1\ \}$ x:=x+1 $\{\ x > 0\ \}$
    - $\{\ y < 2.5\ \}$ y:=2*y $\{\ y < 5\ \}$
    - $\{\ x < 0\ \}$ x:=x+y $\{\ y > x\ \}$

Verification Conditions

- Weakest Precondition
  - What is the precondition that must hold before the statement to ensure that the postcondition holds afterwards?
    - $\{\ x > -1\ \}\ x:=x+1\ \{\ x > 0\ \}$
    - $\{\ y < 2.5\ \}\ y:=2*y\ \{\ y < 5\ \}$
    - $\{\ x < 0\ \}\ x:=x+y\ \{\ y > x\ \}$
    - $\{\qquad\quad\}$ assume $a \leq 5\ \{\ a \leq 5\ \}$
    - $\{\qquad\quad\}$ assume $a \leq b\{\ a \leq 5\ \}$

Verification Conditions

- Weakest Precondition
  - What is the precondition that must hold before the statement to ensure that the postcondition holds afterwards?
    - $\{ x > -1 \}$ x:=x+1 $\{ x > 0 \}$
    - $\{ y < 2.5 \}$ y:=2*y $\{ y < 5 \}$
    - $\{ x < 0 \}$ x:=x+y $\{ y > x \}$
    - $\{$ True $\}$ assume $a \leq 5$ $\{ a \leq 5 \}$
    - $\{ b \leq 5 \}$ assume $a \leq b$ $\{ a \leq 5 \}$

Verification Conditions

- Weakest Precondition
  - What is the precondition that must hold before the statement to ensure that the postcondition holds afterwards?
    - $\{ x > -1 \}\ x{:=}x{+}1\ \{ x > 0 \}$
    - $\{ y < 2.5 \}\ y{:=}2{*}y\ \{ y < 5 \}$
    - $\{ x < 0 \}\ x{:=}x{+}y\ \{ y > x \}$
    - $\{ \text{True} \}$ assume $a \leq 5\ \{ a \leq 5 \}$
    - $\{ b \leq 5 \}$ assume $a \leq b \{ a \leq 5 \}$
  - The weakest precondition is the most general one among valid preconditions. Weakest precondition transformer:

$$\text{wp: FOL} \times \text{stmts} \rightarrow \text{FOL}$$

Weakest Precondition Transformer

- Weakest precondition $\mathrm{wp}(F, S)$ for statements S of basic paths:
  - Assumption: What must hold before statement **assume** c is executed to ensure that F holds afterwards? **If** $c \rightarrow F$ holds before, then satisfying c guarantees that F holds afterwards:

$$\mathbf{wp}(F, \textbf{assume } c) \Leftrightarrow c \rightarrow F$$

$$ex)\ \mathbf{wp}(a \leq 5, \textbf{assume } a \leq 5) \Leftrightarrow (a \leq 5 \rightarrow a \leq 5) \Leftrightarrow \mathbf{True}$$

  - Assignment: What must hold before statement $v := e$ is executed to ensure that $F[\mathrm{v}]$ holds afterward? If $F[\mathrm{e}]$ holds before, then assigning e to v makes $F[\mathrm{v}]$ holds afterward:

$$\mathbf{wp}(F[v], v := e) \Leftrightarrow F[\mathrm{e}]$$

$$ex)\ \mathbf{wp}(\mathrm{x} > 0, \mathrm{x}{:=}\mathrm{x}{+}1) \Leftrightarrow (\mathrm{x}{+}1 > 0) \Leftrightarrow (\mathrm{x} > -1)$$

Weakest Precondition Transformer

- Weakest precondition $\mathrm{wp}(F, S)$ for statements S of basic paths:
  - Assumption:

$$\mathbf{wp}(F, \textit{assume } c) \Leftrightarrow c \rightarrow F$$

  - Assignment:

$$\mathbf{wp}(F[v], v := e) \Leftrightarrow F[\mathrm{e}]$$

  - For a sequence of statements $S_1; \ldots; S_n$, define as:

$$\mathbf{wp}(F, S_1; \ldots; S_n) \Leftrightarrow \mathbf{wp}(\mathbf{wp}(F, S_n), S_1; \ldots; S_{n-1})$$

The weakest precondition moves a formula backward over a sequence of statements.

Verification Conditions

- The verification condition (VC) of basic path

$$
\begin{array}{l}
@F \\
S_1; \\
\vdots \\
S_n; \\
@G
\end{array}
$$

is

$$F \rightarrow \mathbf{wp}(G, S_1; \ldots; S_n).$$

- The VC is sometimes denoted by the Hoare triple

$$\{F\}\ S_1; \ldots; S_n\ \{G\}.$$

Example

- The VC of the basic path

$$\begin{array}{l} @x \geq 0 \\ x := x + 1; \\ @x \geq 1 \end{array}$$

is

$$x \geq 0 \rightarrow \textbf{wp}(x \geq 1, x := x + 1)$$

where

$$\textbf{wp}(x \geq 1, x := x + 1) \iff x \geq 0$$

Example (2)

- Consider the basic path (2) in the LinearSearch example:

$$
\begin{aligned}
&@L : F : l \leq i \wedge (\forall j.\ l \leq j < i \rightarrow a[j] \neq e)\\
&S_1 : \text{assume } i \leq u;\\
&S_2 : \text{assume } a[i] = e;\\
&S_3 : rv := \text{true}\\
&@\text{post } G : rv \leftrightarrow \exists i. l \leq i \leq u \wedge a[i] = e
\end{aligned}
$$

- The VC is $F \rightarrow \text{wp}(G; S_1; S_2; S_3)$, so compute

$$
\begin{aligned}
&\text{wp}(G,\ S_1; S_2; S_3)\\
\Leftrightarrow\ &\text{wp}(\text{wp}(rv \ \leftrightarrow\ \exists j.\ \ell \leq j \leq u\ \wedge\ a[j] = e,\ rv := \texttt{true}),\ S_1; S_2)\\
\Leftrightarrow\ &\text{wp}(\texttt{true}\ \leftrightarrow\ \exists j.\ \ell \leq j \leq u\ \wedge\ a[j] = e,\ S_1; S_2)\\
\Leftrightarrow\ &\text{wp}(\exists j.\ \ell \leq j \leq u\ \wedge\ a[j] = e,\ S_1; S_2)\\
\Leftrightarrow\ &\text{wp}(\text{wp}(\exists j.\ \ell \leq j \leq u\ \wedge\ a[j] = e,\ \texttt{assume}\ a[i] = e),\ S_1)\\
\Leftrightarrow\ &\text{wp}(a[i] = e\ \rightarrow\ \exists j.\ \ell \leq j \leq u\ \wedge\ a[j] = e,\ S_1)\\
\Leftrightarrow\ &\text{wp}(a[i] = e\ \rightarrow\ \exists j.\ \ell \leq j \leq u\ \wedge\ a[j] = e,\ \texttt{assume}\ i \leq u)\\
\Leftrightarrow\ &i \leq u\ \rightarrow\ (a[i] = e\ \rightarrow\ \exists j.\ \ell \leq j \leq u\ \wedge\ a[j] = e)
\end{aligned}
$$

The VC is $l \leq i \wedge (\forall j.\ l \leq j < i \rightarrow a[j] \neq e)$
$\rightarrow (i \leq u \rightarrow (a[i] = e \rightarrow \exists j. l \leq j \leq u \wedge a[j] = e))$

Partial Correctness

Theorem

If for every basic path

$$\begin{array}{l} @F \\ S_1; \\ \vdots \\ S_n; \\ @G \end{array}$$

of program P, the verification condition

$$\{F\}S_1; \ldots; S_n\{G\}$$

is valid, then the program obeys its specification.

Summary

- Inductive assertion method for proving partial correctness.

```
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
}
```

Program

```
@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|
    if (a[i] = e) return true
    i := i + 1;
  }
  return false
}
```

Step1: Generating program annotations.

```
(1)
@pre : 0 ≤ l ∧ u < |a|
i := l;
@L : l ≤ i ∧ (∀j. l ≤ j < i → a[j] ≠ e)

(2)
@L : l ≤ i ∧ (∀j. l ≤ j < i → a[j] ≠ e)
assume i ≤ u;
assume a[i] = e;
rv := true
@post : rv ↔ ∃i.l ≤ i ≤ u ∧ a[i] = e

(3)
@L : l ≤ i ∧ (∀j. l ≤ j < i → a[j] ≠ e)
assume i ≤ u;
assume a[i] ≠ e
i := i + 1;
@L : l ≤ i ∧ (∀j. l ≤ j < i → a[j] ≠ e)

(4)
@L : l ≤ i ∧ (∀j. l ≤ j < i → a[j] ≠ e)
assume i > u;
rv := false
@post : rv ↔ ∃i.l ≤ i ≤ u ∧ a[i] = e
```

Step2: Generating basic paths.

$$
\begin{aligned}
&\mathsf{wp}(G,\ S_1; S_2; S_3)\\
\Leftrightarrow\ &\mathsf{wp}(\mathsf{wp}(rv\ \leftrightarrow\ \exists j.\ \ell \le j \le u\ \wedge\ a[j] = e,\ rv := \mathtt{true}),\ S_1; S_2)\\
\Leftrightarrow\ &\mathsf{wp}(\mathtt{true}\ \leftrightarrow\ \exists j.\ \ell \le j \le u\ \wedge\ a[j] = e,\ S_1; S_2)\\
\Leftrightarrow\ &\mathsf{wp}(\exists j.\ \ell \le j \le u\ \wedge\ a[j] = e,\ S_1; S_2)\\
\Leftrightarrow\ &\mathsf{wp}(\mathsf{wp}(\exists j.\ \ell \le j \le u\ \wedge\ a[j] = e,\ \mathtt{assume}\ a[i] = e),\ S_1)\\
\Leftrightarrow\ &\mathsf{wp}(a[i] = e\ \rightarrow\ \exists j.\ \ell \le j \le u\ \wedge\ a[j] = e,\ S_1)\\
\Leftrightarrow\ &\mathsf{wp}(a[i] = e\ \rightarrow\ \exists j.\ \ell \le j \le u\ \wedge\ a[j] = e,\ \mathtt{assume}\ i \le u)\\
\Leftrightarrow\ &i \le u\ \rightarrow\ (a[i] = e\ \rightarrow\ \exists j.\ \ell \le j \le u\ \wedge\ a[j] = e)
\end{aligned}
$$

The VC is $l \le i \wedge (\forall j.\ l \le j < i \rightarrow a[j] \ne e) \rightarrow (i \le u \rightarrow (a[i] = e \rightarrow \exists j.l \le j \le u \wedge a[j] = e))$

Step3: Generating verification conditions (VC).

Step4:
Checking that VCs are valid.