Generating Verification Conditions for Proving Absence of Run-time Exceptions and Partial Correctness
Objectives

• Understand theory behind proof:
  – Absence of run-time errors
  – Partial correctness

• Understand how verification conditions (proof obligations) are generated

• Insight needed to figure out what contracts need to assert
Outline

• Straight-line code
• Alternation (if statements)
• Iteration
• Procedure calls
• Updating structures (arrays, records)
• Checks involving quantification
Verification Conditions for RTEs & PC

• Prover generates verification conditions (VCs) at locations
  – where compiler checks for a run-time error (RTE)
  – where an assertion must be checked for partial correctness (PC), e.g. post condition at end of module, pre condition at call site, loop invariant, etc.

• VCs are proof obligations – guarantee no RTE or that an assertion holds (when and if control reaches the assertion)

• Prover attempts to reduce the VCs to True.
  – Discharge the proof obligation.
  – Many are discharged automatically (or reduced to False!)
  – Others:
    • Might be too complex for the prover to discharge
    • Might require domain knowledge to be proven
    • Might not be true (the runtime error might occur)
When not all VCs can be discharged

- May need to “strengthen” the program, e.g., add stronger typing constraints, correct error in the code.
- May need to “strengthen” the contracts, e.g., add pre-conditions or intermediate assertions
- May need to add domain knowledge.
- May need to resort to other means: prove VCs outside of SPARK
- Other?
Verification Conditions and Axioms

• SPARK assertions used for proofs include:
  – Pre/post conditions, loop invariants, loop variants, assert conditions, (implicit) assumption that “inputs” are in range
  – (Implicit) check conditions prior to evaluating expressions

• For each path $P$ from a pre/assert/assume condition, $H$, to a “check” condition, $C$:
  – “Hoist” $C$ to the start of $P$, producing $WP(C, P)$ the weakest precondition for $P$ that guarantees $C$
  – Use $H$ as the hypothesis, generating the VC: $H \rightarrow WP(C, P)$

• To prove the VCs, also need axioms expressing type information and semantics of primitive operations
Example: Axioms
May be generated from a description of the target machine

```plaintext
package Standard is
    type Integer is range -2147483648 .. 2147483647;
    type Float is digits 6 range -3.40282E+38..3.40282E+38;
end Standard;

package System is
    type Address is private;
    Min_Int : constant := -9223372036854775808;
    Max_Int : constant := 9223372036854775807;
    Max_Binary_Modulus : constant := 18446744073709551615 + 1;
    Max_Digits : constant := 18;
    Max_Base_Digits : constant := 18;
    Max_Mantissa : constant := 63;
    Storage_Unit : constant := 8;
    Word_Size : constant := 32;
    integer_first = -2147483648
    integer_last = 2147483647
    integer_width = ...
end System;
```
Ex : Axioms generated from type declarations

```plaintext
subtype Value_Type is Integer range 0 .. 10;
subtype Sum_Type is Integer range 0 .. 20;

procedure Sum_Two (P1, P2 : in Value_Type;
                    S : out Sum_Type)
is begin
    S := P1;
    S := S + P2;
end Sum_Two
```

value_type_first = 0
value_type_last = 10
sum_type_first = 0
sum_type_last = 20

Prover uses axioms as hypotheses in discharging the VCs (i.e., reducing them to True).
Example: Implicit assumptions and checks

```pascal
subtype Value_Type is Integer range 0 .. 10;
subtype Sum_Type is Integer range 0 .. 20;

procedure Sum_Two (P1, P2 : in Value_Type;
                   S : out Sum_Type)
is begin
  S := P1;
  S := S + P2;
end Sum_Two;

Assume: Value_Type'First <= P1 and
        P1 <= Value_Type'Last and
        Value_Type'First <= P2 and
        P2 <= Value_Type'Last

Check1: Sum_Type'First <= P1 and
        P1 <= Sum_Type'Last

Check2: Sum_Type'First <= S + P2
        and S + P2 <= Sum_Type'Last
```
WP: “Hoisting” past straight-line code

\[ WP(C, \text{“} X := \text{exp} \text{”}) = C[X/exp] \]

\[ WP(C, \text{“} s1; s2 \text{”}) = WP(WP(C, \text{“} s2 \text{”}), \text{“} s1 \text{”}) \]

\[ WP(C, \text{“} \text{assume/assert/pre H} \text{”}) = H \rightarrow C \]
Example: Generating VC for Check1

```haskell
subtype Value_Type is Integer range 0 .. 10;
subtype Sum_Type is Integer range 0 .. 20;

procedure Sum_Two (P1, P2 : in Value_Type;
                    S : out Sum_Type)
is begin
  S := P1;
  S := S + P2;
end Sum_Two;

Assume: Value_Type'First <= P1 and
        P1 <= Value_Type'Last and
        Value_Type'First <= P2 and
        P2 <= Value_Type'Last

⇒

value_type__first \leq p1 \leq value_type_last \land
value_type_first \leq p2 \leq value_type_last \land

⇒

sum_type__first \leq p1 \leq sum_type_last
```

VC for check1
Example: Generating the VC for Check2

```vhdl
subtype Value_Type is Integer range 0 .. 10;
subtype Sum_Type is Integer range 0 .. 20;

procedure Sum_Two (P1, P2 : in Value_Type;
                    S : out Sum_Type)
is begin
    S := P1;
    S := S + P2;
end Sum_Two;
```

```
Sum_Type’First <= S + P2 and
S + P2 <= Sum_Type’Last
```
Example: Generating the VC for Check2

```plaintext
subtype Value_Type is Integer range 0 .. 10;
subtype Sum_Type is Integer range 0 .. 20;

procedure Sum_Two (P1, P2 : in Value_Type;
                   S : out Sum_Type)
is begin
  S := P1;
  S := S + P2;
end Sum_Two;
```

Sum_Type'First <= P1 + P2 and
P1 + P2 <= Sum_Type'Last
Example: Generating the VC for Check2

```
subtype Value_Type is Integer range 0 .. 10;
subtype Sum_Type is Integer range 0 .. 20;

procedure Sum_Two (P1, P2 : in Value_Type;
                    S : out Sum_Type)
is begin
  S := P1;
  S := S + P2;
end Sum_Two;
```

Assume: Value_Type'First <= P1 and
        P1 <= Value_Type'Last and
        Value_Type'First <= P2 and
        P2 <= Value_Type'Last

```
value_type__first \leq p1 \leq value_type_last \land
value_type__first \leq p2 \leq value_type_last \land
\rightarrow
sum_type__first \leq p1 + p2 \leq sum_type_last
```

VC for check2
Example: assumptions and checks

procedure Add_Threshold (X: in Amount) with
Pre => Total <= Amount'Last,
Post =>
   ((X + Total'Old <= Threshold and Total = Total'Old + X)
    or (X + Total'Old > Threshold and Total = Threshold));

Assume: Amount'First <= X …
   Total >= Integer'First …
Pre:  Total <= Amount'Last

Check: Integer'First <= Total + X …

Post: (X + Total'Old <= Threshold
   and Total = Total'Old + X)
   or (X + Total'Old > Threshold
   and Total = Threshold)
WP: “Hoisting” constraints

\[ WP(C, "X := exp") = C[X/exp] \]
\[ WP(C, "s1 ; s2") = WP(WP(C, "s2"), "s1") \]
\[ WP(C, "assume/assert H") = H \rightarrow C \]
\[ WP(C, "if b then") = b \rightarrow C \]
\[ WP(C, "if b-else") = (\neg b) \rightarrow C \]

When hoisting a check past “if b ...”

assert \textit{b} on the \textit{then} branch (the \textit{True} branch)
assert \textit{not b} on the \textit{else} branch (the \textit{False} branch)
Example: Generating VC for Check

```
procedure Add_Threshold (X: in Amount) is begin
  if Total + X <= Threshold then
    Total := Total + X;
  else
    Total := Threshold;
  end if;
end Add_Threshold;
```

```
Integer'First <= Total + X and
Total + X <= Integer'Last
```
Example: Generating VC for Check

procedure Add_Threshold (X: in Amount) is begin
    if Total + X <= Threshold then
        Total := Total + X;
    else
        Total := Threshold;
    end if;
end Add_Threshold;

Total + X <= Threshold →
Integer'First <= Total + X and Total + X <= Integer'Last
**Example: Generating VC for Check**

```plaintext
procedure Add_Threshold (X: in Amount) is begin
  if Total + X <= Threshold then
    Total := Total + X;
  else
    Total := Threshold;
  end if;
end Add_Threshold;
```

**Assume:**
- Amount'First <= X ...
- Total >= Integer'First ...
- Total <= Amount'Last

Total + X <= Threshold

⇒

Integer'First <= Total + X and
Total + X <= Integer'Last
Example: Generating VC for Check

```
procedure Add_Threshold (X: in Amount) is
  begin
    if Total + X <= Threshold then
      Total := Total + X;
    else
      Total := Threshold;
    end if;
  end Add_Threshold;
```

Assume: Amount’First <= X ...
Total >= Integer’First ...
Pre: Total <= Amount’Last

\[
\text{Total + X <= Threshold}
\rightarrow
\text{Integer’First <= Total + X and Total + X <= Integer’Last}
\]

\[
\begin{align*}
\text{amount_first} & \leq x \ldots \land \\
\text{total} & \geq \text{integer_first} \ldots \land \\
\text{total} & \leq \text{amount_last} \land \\
\text{total + x} & \leq \text{threshold} \\
\rightarrow \\
\text{integer_first} & \leq \text{total + x} \land \text{total + x} \leq \text{integer_last}
\end{align*}
\]

VC for RTE check on the path that takes the then branch
Example: Generating VCs for Post

For each path \( P \) from the start to the finish, after hoisting "Post" to the start of \( P \), replace each \( V' \text{Old} \) with \( V \), to get \( WP(\text{"Post"}, P) \)

Then generate VC: \( H \rightarrow WP(\text{"Post"}, P) \)
Example: Generating VCs for Post

procedure Add_Threshold (X: in Amount) is begin
  if Total + X <= Threshold then
    Total := Total + X;
  else
    Total := Threshold;
  end if;
end Add_Threshold;

(X + Total'Old <= Threshold
  and Total = Total'Old + X)
or (X + Total'Old > Threshold
  and Total = Threshold)
Example: Generating VCs for Post

procedure Add_Threshold (X: in Amount) is begin
  if Total + X <= Threshold then
    Total := Total + X;
  else
    Total := Threshold;
  end if;
end Add_Threshold;

(X + Total'Old <= Threshold
  and Threshold = Total'Old + X)
or (X + Total'Old > Threshold
  and Threshold = Threshold)
Example: Generating VCs for Post

procedure Add_Threshold (X: in Amount) is begin
  if Total + X <= Threshold then
    Total := Total + X;
  else
    Total := Threshold;
  end if;
end Add_Threshold;

\[\text{not}(\text{Total} + X \leq \text{Threshold}) \rightarrow \]
\[(X + \text{Total}'\text{Old} \leq \text{Threshold} \quad \text{and} \quad \text{Threshold} = \text{Total}'\text{Old} + X) \quad \text{or} \quad (X + \text{Total}'\text{Old} > \text{Threshold} \quad \text{and} \quad \text{Threshold} = \text{Threshold})\]
Example: Generating VCs for Post

```vhdl
procedure Add_Threshold (X: in Amount) is begin
    if Total + X <= Threshold then
        Total := Total + X;
    else
        Total := Threshold;
    end if;
end Add_Threshold;
```

\[\neg (\text{Total} + X \leq \text{Threshold})\]

\[\rightarrow\]

\[(X + \text{Total} \leq \text{Threshold}
\quad\text{and}\quad\text{Threshold} = \text{Total} + X)\]

\[\text{or}\quad(X + \text{Total} > \text{Threshold}
\quad\text{and}\quad\text{Threshold} = \text{Threshold})\]
Example: Generating VCs for Post

procedure Add_Threshold (X: in Amount) is begin
  if Total + X <= Threshold then
    Total := Total + X;
  else
    Total := Threshold;
  end if;
end Add_Threshold;

Assume: Amount'First <= X ... Total >= Integer'First ...

Pre: Total <= Amount'Last

not(Total + X <= Threshold) →
(X + Total <= Threshold and Threshold = Total + X) or (X + Total > Threshold and Threshold = Threshold)

VC for post-condition on the path that takes the else branch

amount_first ≤ x ... ∧
total ≥ integer_first ... ∧
total ≤ amount_last ∧
total + x ≠ threshold) →
(x + total ≤ threshold ∧ threshold = total + x) ∨
(x + total > threshold ∧ threshold = threshold)
Example: Generating VCs for Post

procedure Add_Threshold (X: in Amount) is begin
  if Total + X ≤ Threshold then
    Total := Total + X;
  else
    Total := Threshold;
  end if;
end Add_Threshold;

(X + Total'Old ≤ Threshold
and Total = Total'Old + X)
or (X + Total'Old > Threshold
and Total = Threshold)
procedure Add_Threshold (X: in Amount) is begin
    if Total + X <= Threshold then
        Total := Total + X;
    else
        Total := Threshold;
    end if;
end Add_Threshold;

(X + Total'Old <= Threshold
    and Total + X = Total'Old + X)
or (X + Total'Old > Threshold
    and Total + X = Threshold)
Example: Generating VCs for Post

```plaintext
procedure Add_Threshold (X: in Amount) is begin
  if Total + X <= Threshold then
    Total := Total + X;
  else
    Total := Threshold;
  end if;
end Add_Threshold;
```

- \( Total + X \leq Threshold \)\
  - \( X + Total'Old \leq Threshold \) and \( Total + X = Total'Old + X \)
  - or \( X + Total'Old > Threshold \) and \( Total + X = Threshold \)
Example: Generating VCs for Post

```vhdl
procedure Add_Threshold (X: in Amount) is begin
    if Total + X <= Threshold then
        Total := Total + X;
    else
        Total := Threshold;
    end if;
end Add_Threshold;
```

\[ Total + X \leq \text{Threshold} \quad \rightarrow \quad (X + Total \leq \text{Threshold} \text{ and } Total + X = Total + X) \text{ or } (X + Total > \text{Threshold} \text{ and } Total + X = \text{Threshold}) \]
Example: Generating VCs for Post

```
procedure Add_Threshold (X: in Amount) is begin
  if Total + X <= Threshold then
    Total := Total + X;
  else
    Total := Threshold;
  end if;
end Add_Threshold;
```

Assume:
- Amount’First <= X ...
- Total >= Integer’First ...
- Total <= Amount’Last

```
Total + X <= Threshold
→
(X + Total <= Threshold
and Total + X = Total + X)
or (X + Total > Threshold
and Total + X = Threshold)
```
Example: Generating VCs for Post

```
procedure Add_Threshold (X: in Amount) is
begin
  if Total + X <= Threshold then
    Total := Total + X;
  else
    Total := Threshold;
  end if;
end Add_Threshold;
```

Assume:
- Amount'First <= X ...
- Total >= Integer'First ...
- Total <= Amount'Last

```
Total + X <= Threshold

→
(X + Total <= Threshold
and Total + X = Total + X)
or (X + Total > Threshold
and Total + X = Threshold)
```

\[ amount \_first \leq x \quad \land \]
\[ \ldots \land \\
\[ total + x \leq threshold \quad \land \\
\[ \rightarrow \\
\[ (x + total \leq threshold \land total + x = total + x) \lor \\
\[ (x + total > threshold \land total + x = threshold) \]
```

VC for post on the path that takes the then branch
Example: Iteration

To prove PC, each loop has to be cut by a loop invariant

```plaintext
procedure Divide(M, N: in Natural;
    Q, R: out Natural)
    with Pre => N > 0,
    Post => M = N * Q + R and R < N;

procedure Divide(M, N: in Natural;
    Q, R: out Natural)
is begin
    R := M;
    Q := 0;
    while R >= N loop
        pragma Loop_Invariant (M = N * Q + R);
        Q := Q + 1;
        R := R - N;
    end loop;
end Divide;
```
Loop Invariant - Semantics

• Dynamic semantics: Like a regular assertion

• Static semantics:
  – Must be provable the first time control reaches it
  – Must be strong enough that assuming only the invariant
    • allows proof that the invariant is true the next time control reaches it
      (i.e., the loop preserves the invariant)
    • allows proof of no RTEs in (or after) the loop
    • allows proof of desired post condition, if any

• Inductively, the loop invariant is true whenever control reaches it
Example: First time control reaches invariant

procedure Divide(M, N: in Natural;
    Q, R: out Natural)
begin
    R := M;
    Q := 0;
    while R >= N loop
        pragma Loop_Invariant (M = N * Q + R);
        Q := Q + 1;
        R := R - N;
    end loop;
end Divide;

Assume: M, N >= 0 and M, N <= ...
Pre:   N > 0

Initialization Check:
M = N * Q + R
Example: First time control reaches invariant

```verbatim
procedure Divide(M, N: in Natural;
    Q, R: out Natural)
is begin
    R := M;
    Q := 0;
    while R >= N loop
        pragma Loop_Invariant (M = N * Q + R);
        Q := Q + 1;
        R := R - N;
    end loop;
end Divide;
```

\[ M = N \times Q + R \]
procedure Divide(M, N: in Natural;
Q, R: out Natural)
is begin
R := M;
Q := 0;
while R >= N loop
pragma Loop_Invariant (M = N * Q + R);
Q := Q + 1;
R := R - N;
end loop;
end Divide;

R >= N →
M = N * Q + R
Example: First time control reaches invariant

procedure Divide(M, N: in Natural;
Q, R: out Natural)

is begin

R := M;
Q := 0;

while R >= N loop

pragma Loop_Invariant (M = N * Q + R);

Q := Q + 1;
R := R - N;
end loop;
end Divide;
Example: First time control reaches invariant

```plaintext
procedure Divide(M, N: in Natural;
                 Q, R: out Natural)
  is begin
    R := M;
    Q := 0;
    while R >= N loop
      pragma Loop_Invariant (M = N * Q + R);
      Q := Q + 1;
      R := R - N;
    end loop;
  end Divide;
```

Example: First time control reaches invariant

```
procedure Divide(M, N: in Natural;
                 Q, R: out Natural)
  is begin
    R := M;
    Q := 0;
    while R >= N loop
      pragma Loop_Invariant (M = N * Q + R);
      Q := Q + 1;
      R := R - N;
    end loop;
  end Divide;
```

Example: First time control reaches invariant

```
procedure Divide(M, N: in Natural;
                 Q, R: out Natural)
  is begin
    R := M;
    Q := 0;
    while R >= N loop
      pragma Loop_Invariant (M = N * Q + R);
      Q := Q + 1;
      R := R - N;
    end loop;
  end Divide;
```

Example: First time control reaches invariant

```
procedure Divide(M, N: in Natural;
                 Q, R: out Natural)
  is begin
    R := M;
    Q := 0;
    while R >= N loop
      pragma Loop_Invariant (M = N * Q + R);
      Q := Q + 1;
      R := R - N;
    end loop;
  end Divide;
```

Example: First time control reaches invariant

```
procedure Divide(M, N: in Natural;
                 Q, R: out Natural)
  is begin
    R := M;
    Q := 0;
    while R >= N loop
      pragma Loop_Invariant (M = N * Q + R);
      Q := Q + 1;
      R := R - N;
    end loop;
  end Divide;
```
Procedure Divide(M, N: in Natural;
    Q, R: out Natural)
is begin
    R := M;
    Q := 0;
    while R >= N loop
        pragma Loop_Invariant (M = N * Q + R);
        Q := Q + 1;
        R := R - N;
    end loop;
end Divide;

Assume: M, N >= 0 and M, N <= ...
Pre: N > 0

M >= N
→
M = N * 0 + M

VC proving that invariant holds the first time control reaches it
Example: “Next” iteration check

```
procedure Divide(M, N: in Natural;
                Q, R: out Natural)
is begin
    R := M;
    Q := 0;
    while R >= N loop
        pragma Loop_Invariant (M = N * Q + R);
        Q := Q + 1;
        R := R - N;
    end loop;
end Divide;
```

Iteration Check:

```
M = N * Q + R
```

Assume inv:

```
M = N * Q + R
```
Example: generating VCs for iteration check

```
procedure Divide(M, N: in Natural;
                Q, R: out Natural)
is begin
    R := M;
    Q := 0;
    while R >= N loop
        pragma Loop_Invariant (M = N * Q + R);
        Q := Q + 1;
        R := R - N;
    end loop;
end Divide;
```

```
M = N * Q + R
```

Example: generating VCs for iteration check

procedure Divide(M, N: in Natural;
                 Q, R: out Natural)
is begin
  R := M;
  Q := 0;
  while R >= N loop
    pragma Loop_Invariant (M = N \* Q + R);
    Q := Q + 1;
    R := R - N;
  end loop;
end Divide;

R >= N
→
M = N \* Q + R
Example: generating VCs for iteration check

```
procedure Divide(M, N: in Natural;
Q, R: out Natural)
is begin
    R := M;
    Q := 0;
    while R >= N loop
        pragma Loop_Invariant (M = N * Q + R);
        Q := Q + 1;
        R := R - N;
    end loop;
end Divide;
```

R - N \geq N \implies M = N \times Q + R - N
Example: generating VCs for iteration check

```haskell
procedure Divide(M, N: in Natural;
    Q, R: out Natural)
is begin
    R := M;
    Q := 0;
    while R >= N loop
        pragma Loop_Invariant (M = N * Q + R);
        Q := Q + 1;
        R := R - N;
    end loop;
end Divide;
```

\[
R - N \geq N \\
\Rightarrow \\
M = N \times (Q + 1) + R - N
\]
Example: generating VCs for iteration check

```plaintext
procedure Divide(M, N: in Natural;
    Q, R: out Natural)
is begin
    R := M;
    Q := 0;
    while R >= N loop
        pragma Loop_Invariant (M = N * Q + R);
        Q := Q + 1;
        R := R - N;
    end loop;
end Divide;
```

Assume inv:

\[
M = N \times Q + R
\]

R - N >= N

\[
M = N \times (Q + 1) + R - N
\]
Example: generating VCs for iteration check

```
procedure Divide(M, N: in Natural;  
                 Q, R: out Natural)
is begin
    R := M;
    Q := 0;
    while R >= N loop
        pragma Loop_Invariant (M = N * Q + R);
        Q := Q + 1;
        R := R - N;
    end loop;
end Divide;
```

Assume inv:

\[ M = N \cdot Q + R \]

```
R - N >= N  
\rightarrow  
M = N \cdot (Q + 1) + R - N
```

VC proving that the loop preserves the invariant

\[ m = n \cdot q + r \land 
   r - n \geq n 
\rightarrow 
   m = n \cdot (q + 1) + r - n \]
Example: Post condition checks & iteration

```plaintext
procedure Divide(M, N: in Natural;
                 Q, R: out Natural)
is begin
  R := M;
  Q := 0;
  while R >= N loop
    -- pragma Loop_Invariant (M = N * Q + R);
    Q := Q + 1;
    R := R - N;
  end loop;
end Divide;
```

**Assume:** $M, N \geq 0$ and $M, N \leq \ldots$

**Pre:** $N > 0$

**Assume inv:**

$$M = N \times Q + R$$

**Post:**

$$M = N \times Q + R \text{ and } R < N$$
Example: generating VCs for PC

```plaintext
procedure Divide(M, N: in Natural;
                 Q, R: out Natural)
  is begin
    R := M;
    Q := 0;
    while R >= N loop
      pragma Loop_Invariant (M = N * Q + R);
      Q := Q + 1;
      R := R - N;
    end loop;
  end Divide;
```

\[ M = N \times Q + R \text{ and } R < N \]

Example: generating VCs for PC

```plaintext
procedure Divide(M, N: in Natural;
        Q, R: out Natural)
is begin
    R := M;
    Q := 0;
    while R >= N loop
        pragma Loop_Invariant (M = N * Q + R);
        Q := Q + 1;
        R := R - N;
    end loop;
end Divide;
```

\[ \text{not} (R \geq N) \rightarrow M = N \times Q + R \text{ and } R < N \]
Example: generating VCs for PC

procedure Divide(M, N: in Natural;
    Q, R: out Natural)
is begin
    R := M;
    Q := 0;
    while R >= N loop
        pragma Loop_Invariant (M = N * Q + R);
        Q := Q + 1;
        R := R - N;
    end loop;
end Divide;

not (R - N >= N) →
M = N * Q + R - N and R - N < N
Example: generating VCs for PC

procedure Divide(M, N: in Natural;
Q, R: out Natural)

is begin
    R := M;
    Q := 0;
    while R >= N loop
        pragma Loop_Invariant (M = N * Q + R);
        Q := Q + 1;
        R := R - N;
    end loop;
end Divide;

not (R - N >= N) →
M = N * (Q + 1) + R - N and R - N < N
Example: generating VCs for PC

```plaintext
procedure Divide(M, N: in Natural;
    Q, R: out Natural)
is begin
    R := M;
    Q := 0;
    while R >= N loop
        pragma Loop_Invariant (M = N * Q + R);
        Q := Q + 1;
        R := R - N;
    end loop;
end Divide;
```

Assume inv:

\[ M = N \times Q + R \]

\[ \text{not } (R - N \geq N) \rightarrow M = N \times (Q + 1) + R - N \text{ and } R - N < N \]

\[ m = n \times q + r \land r - n \neq n \rightarrow m = n \times (q + 1) + r - n \land r - n < n \]

VC showing that the invariant guarantees the post condition
Example: generating VCs for PC

```
procedure Divide(M, N: in Natural;
                 Q, R: out Natural)
is begin
  R := M;
  Q := 0;
  while R >= N loop
    pragma Loop_Invariant (M = N * Q + R);
    Q := Q + 1;
    R := R - N;
  end loop;
end Divide;
```

\[ M = N \times Q + R \text{ and } R < N \]
Example: generating VCs for PC

```
procedure Divide(M, N: in Natural;
                 Q, R: out Natural)
is begin
  R := M;
  Q := 0;
  while R >= N loop
    pragma Loop_Invariant (M = N * Q + R);
    Q := Q + 1;
    R := R - N;
  end loop;
end Divide;
```

not (R >= N) → M = N * Q + R and R < N
Example: generating VCs for PC

```plaintext
procedure Divide(M, N: in Natural;
                 Q, R: out Natural)
is begin
    R := M;
    Q := 0;
    while R >= N loop
        pragma Loop_Invariant (M = N * Q + R);
        Q := Q + 1;
        R := R - N;
    end loop;
end Divide;
```

not (R >= N) →

M = N * 0 + R and R < N
Example: generating VCs for PC

procedure Divide(M, N: in Natural;
    Q, R: out Natural)
is begin
    R := M;
    Q := 0;
    while R >= N loop
        pragma Loop_Invariant (M = N * Q + R);
        Q := Q + 1;
        R := R - N;
    end loop;
end Divide;

not (M >= N) →
M = N * 0 + M and M < N
Example: generating VCs for PC

procedure Divide(M, N: in Natural;
Q, R: out Natural)
is begin
  R := M;
  Q := 0;
  while R >= N loop
    pragma Loop_Invariant (M = N * Q + R);
    Q := Q + 1;
    R := R - N;
  end loop;
end Divide;

Assume: M, N >= 0 and M, N <= ...
Pre: N > 0

not (M >= N) →
M = N * 0 + M and M < N

VC showing post condition holds if the loop is never entered

... ∧
n > 0 ∧
m ≠ n
→
m = n * 0 + m ∧
m < n
Procedure call in a Client

procedure P (X: in,
       Y: in out,
       Z: out)
with Global => (IN_OUT=>W),
Pre => PRE,
Post => POST;

procedure Q (...) is
begin
  . . .
  P(exp, A, B);
  . . .
end Q;

• Must check PRE[X/exp, Y/A] at the call site.

• Hoist over the call (“new” variables denote exported values):

\[
WP(C, "P(exp, A, B)") = \]
PRE[X/exp, Y/A] and
POST[X/exp, Y'Old/A, Y/A_N, Z/B_N, W'Old/W, W/W_N] \rightarrow C[A/A_N, B/B_N, W/W_N]
procedure Add_Threshold (Y: in Amount)
with Global => (In_Out => Total),
  Pre => Total <= Amount'Last,
  Contract_Cases =>
    (Y + Total <= Threshold => Total = Total'Old + Y,
     Y + Total > Threshold => Total = Threshold);

procedure Add (X: in Amount)
with Global => (In_Out => Total),
  Pre => Total <= Amount'Last,
  Post => Total = Total'Old + X;

procedure Add_Threshold (Y: in Amount) is begin
  if Total + Y <= Threshold then
    Add(Y);
  else
    Total := Threshold;
  end if;
end Add_Threshold;
Aside: Contract Cases

Contract cases is a shorthand:

```plaintext
procedure P (...) with Pre => PRE, Post => POST, Contract_Cases => (A1 => B1, ..., An => Bn);
```

Expands to:

```plaintext
procedure P (...) with Pre => PRE
    and then Exactly_One_Of (A1, ..., An),
Post => POST
    and then (if A1'Old then B1)
    ...
    and then (if An'Old then Bn);
```
Example: Procedure call in a Client

procedure Add_Threshold (Y: in Amount)
with Global => (In_Out => Total),
  Pre => Total <= Amount'Last and then Exactly_One_Of...
  Post =>
    (if Y + Total'Old <= Threshold then
      Total = Total'Old + Y
      and then if Y + Total'Old > Threshold =>
      Total = Threshold);

Note:
Exactly_One_Of(X + Total <= Threshold, X + Total > Threshold)
is True.

Because of this, it does not affect the PC proof; to simplify the following slides, I will therefore ignore it in the sequel.
Example: assumptions and checks

procedure Add (X: in Amount) with Global => (In_Out => Total),
Pre => Total <= Amount'Last,
Post => Total = Total'Old + X;

procedure Add_Threshold (Y: in Amount) is begin
    if Total + Y <= Threshold then
        Add(Y);
    else
        Total := Threshold;
    end if;
end Add_Threshold;

Assume: Amount'First <= Y …
Total >= Integer'First …
Pre: Total <= Amount'Last

Check Post:
(if Y + Total'Old <= Threshold then
    Total = Total'Old + Y
and then if Y + Total'Old > Threshold =>
    Total = Threshold);

Check PRE_{Add}:
Total <= Amount'Last
Example: generating VC at call site

procedure Add (X: in Amount)
with Global => (In_Out => Total),
    Pre => Total <= Amount'Last,
    Post => Total = Total'Old + X;

procedure Add_Threshold (Y: in Amount) is begin
    if Total + Y <= Threshold then
        Add(Y);
    else
        Total := Threshold;
    end if;
end Add_Threshold;
Example: generating VC at call site

```haskell
procedure Add (X: in Amount)
  with Global => (In_Out => Total),
  Pre => Total <= Amount'Last,
  Post => Total = Total'Old + X;
```

```haskell
procedure Add_Threshold (Y: in Amount) is begin
  if Total + Y <= Threshold then
    Add(Y);
  else
    Total := Threshold;
  end if;
end Add_Threshold;
```

Total + Y <= Threshold

Total <= Amount'Last
Example: generating VC at call site

```plaintext
procedure Add (X: in Amount) with Global => (In_Out => Total),
Pre => Total <= Amount'Last,
Post => Total = Total'Old + X;

procedure Add_Threshold (Y: in Amount) is begin
  if Total + Y <= Threshold then
    Add(Y);
  else
    Total := Threshold;
  end if;
end Add_Threshold;
```

Assume:
- \( \text{Amount'}\text{First} \leq Y \) ...
- \( \text{Total} \geq \text{Integer'}\text{First} \) ...

Pre:
- \( \text{Total} \leq \text{Amount'}\text{Last} \)

\[ \text{Total + Y} \leq \text{Threshold} \rightarrow \text{Total} \leq \text{Amount'}\text{Last} \]

\[ \ldots \land \text{total} \leq \text{amount}_\text{last} \land \text{total} + y \leq \text{threshold} \land \rightarrow \text{total} \leq \text{amount}_\text{last} \]

VC showing called procedure's pre-condition holds at the call site
Example: generating VCs for PC

procedure Add (X: in Amount) with Global => (In_Out => Total),
  Pre => Total <= Amount’Last,
  Post => Total = Total’Old + X;

procedure Add_Threshold (Y: in Amount) is begin
  if Total + Y <= Threshold then
    Add(Y);
  else
    Total := Threshold;
  end if;
end Add_Threshold;

(if Y + Total’Old <= Threshold then
  Total = Total’Old + Y
and then if Y + Total’Old > Threshold =>
  Total = Threshold);
Example: generating VCs for PC

procedure Add (X: in Amount)
with Global => (In_Out => Total),
  Pre => Total <= Amount'Last,
  Post => Total = Total'Old + X;

procedure Add_Threshold (Y: in Amount) is begin
  if Total + Y <= Threshold then
    Add(Y);
  else
    Total := Threshold;
  end if;
end Add_Threshold;

(if Y + Total'Old <= Threshold then
  Total = Total'Old + Y
and then if Y + Total'Old > Threshold =>
  Total = Threshold);
Example: assumptions and checks

procedure Add (X: in Amount)
with Global => (In_Out => Total),
  Pre => Total <= Amount'Last,
  Post => Total = Total'Old + X;

procedure Add_Threshold (Y: in Amount) is begin
  if Total + Y <= Threshold then
    Add(Y);
  else
    Total := Threshold;
  end if;
end Add_Threshold;

Total <= Amount'Last and
Total₁ = Total + Y
→
(if Y + Total'Old <= Threshold then
  Total₁ = Total'Old + Y
and then if Y + Total'Old > Threshold =>
  Total₁ = Threshold)
procedure Add_Threshold (Y: in Amount) is begin
  if Total + Y <= Threshold then
    Add(Y);
  else
    Total := Threshold;
  end if;
end Add_Threshold;

Total + Y <= Threshold
Total <= Amount’Last and
Total₁ = Total + Y
→
(if Y + Total’Old <= Threshold then
  Total₁ = Total’Old + Y
  and then if Y + Total’Old > Threshold =>
  Total₁ = Threshold)
Example: generating VCs for PC

```haskell
procedure Add_Threshold (Y: in Amount) is begin
    if Total + Y <= Threshold then
        Add(Y);
    else
        Total := Threshold;
    end if;
end Add_Threshold;
```

Total + Y <= Threshold
Total <= Amount’Last and
Total₁ = Total + Y
→
(if Y + Total <= Threshold then
    Total₁ = Total + Y
and then if Y + Total > Threshold =>
    Total₁ = Threshold)
Example: generating VCs for PC

```vhdl
procedure Add_Threshold (Y: in Amount) is
begin
  if Total + Y <= Threshold then
    Add(Y);
  else
    Total := Threshold;
  end if;
end Add_Threshold;
```

VC for post-condition on the path that takes the then branch

Assume: Amount'First <= Y ...

Pre: Total <= Amount'Last

Total + Y <= Threshold
Total <= Amount'Last and
Total₁ = Total + Y →
(if Y + Total <= Threshold then
Total₁ = Total + Y
and then if Y + Total > Threshold =>
Total₁ = Threshold)
Example: Updating structures

```plaintext
procedure Swap (I, J: in Index;
   A: in out ArrType)
   with Post =>
      A = A'Old'Update(I => A'Old(J),
                        J => A'Old(I));
```

```plaintext
procedure Swap (I, J: in Index;
   A: in out ArrType)
   is
      T: integer;
   begin
      T := A(I);
      A(I) := A(J);
      A(J) := T;
   end Swap;
```

“Hoist” over indexed assignment:

\[ WP(C, "A(\text{exp1}) := \text{exp2}\") = C[A/A'Update(\text{exp1} => \text{exp2})] \]
Example: Updating structures

procedure Swap (I, J: in Index; A: in out ArrType) is
    T: integer;
begin
    T := A(I);
    A(I) := A(J);
    A(J) := T;
end Swap;

A = A'Old'Update (J => A'Old(I),
                 I => A'Old(J))
Example: Updating structures

procedure Swap (I, J: in Index; A: in out ArrType)
   is
     T: integer;
   begin
     T := A(I);
     A(I) := A(J);
     A(J) := T;
   end Swap;

A'Update(J => T) =
A'Old'Update(J => A'Old(I),
              I => A'Old(J))
Example: Updating structures

```haskell
procedure Swap (I, J: in Index; A: in out ArrType) is
  T: integer;
begin
  T := A(I);
  A(I) := A(J);
  A(J) := T;
end Swap;
```

\[
A'\text{Update}(I \Rightarrow A(J))'\text{Update}(J \Rightarrow T) = \text{A'Old'Update}(J \Rightarrow \text{A'Old}(I), I \Rightarrow \text{A'Old}(J))
\]
Example: Updating structures

procedure Swap (I, J: in Index; A: in out ArrType) is
begin
  T := A(I);
  A(I) := A(J);
  A(J) := T;
end Swap;

A'Update(I => A(J))'Update(J => A(I)) = A'Old'Update(J => A'Old(I),
  I => A'Old(J))
Example: Updating structures

procedure Swap (I, J: in Index; A: in out ArrType) is
begin
  T := A(I);
  A(I) := A(J);
  A(J) := T;
end Swap;

A'Update(I => A(J))'Update(J => A(I)) =
  A'Update(J => A(I),
            I => A(J))
Example: Updating structures

procedure Swap (I, J: in Index; A: in out ArrType)
    is
        T: integer;
    begin
        T := A(I);
        A(I) := A(J);
        A(J) := T;
    end Swap;

update( update( a, i, a(j) ), j, a(i) )
    =
    update( update( a, j, a(i) ), i, a(j) )

VC for procedure post-condition
Aside: Axioms/rules define update

\[
\text{update}( a, i, t ) (i) = t \\
i' \neq i \implies \text{update}( a, i, t ) (i') = a(i')
\]

Case \( i \neq j \):

• For index \( i \):

\[
\text{update}( \text{update}( a, i, a(j) ), j, a(i) ) (i) = \text{update}( a, i, a(j) ) (i) = a(j) \\
\text{update}( \text{update}( a, j, a(i) ), i, a(j) ) (i) = a(j)
\]

• For index \( j \):

\[
\text{update}( \text{update}( a, i, a(j) ), j, a(i) ) (j) = a(i) \\
\text{update}( \text{update}( a, j, a(i) ), i, a(j) ) (i) = \text{update}( a, j, a(i) ) (j) = a(i)
\]

• For any other index \( k \):

\[
\text{update}( \text{update}( a, i, a(j) ), j, a(i) ) (k) = \text{update}( a, i, a(j) ) (k) = a(k) \\
\text{update}( \text{update}( a, j, a(i) ), i, a(j) ) (k) = \text{update}( a, j, a(i) ) (k) = a(k)
\]
Aside: Axioms/rules define update

\[
\text{update}(a, i, t)(i) = t \\
i' \neq i \rightarrow \text{update}(a, i, t)(i') = a(i')
\]

Case \(i = j\):

• For index \(i\):

\[
\begin{align*}
\text{update}\left(\text{update}(a, i, a(j)), j, a(i)\right)(i) \\
= & \text{update}\left(\text{update}(a, i, a(j)), j, a(i)\right)(j) = a(i) \\
\text{update}\left(\text{update}(a, j, a(i)), i, a(j)\right)(i) = a(j) = a(i)
\end{align*}
\]

• For any other index \(k\):

\[
\begin{align*}
\text{update}\left(\text{update}(a, i, a(j)), j, a(i)\right)(k) = & \text{update}(a, i, a(j))(k) = a(k) \\
\text{update}\left(\text{update}(a, j, a(i)), i, a(j)\right)(k) = & \text{update}(a, j, a(i))(k) = a(k)
\end{align*}
\]

So VC can be reduced to True
Example: Quantified conditions

function Max_Index (A : T_Arr) return Positive with
Pre  => (A'Length > 0),
Post =>
((for all J in A'Range => A (J) <= A (Max_Index'Result)) and
(for all J in A'First .. Max_Index'Result - 1 =>
 A (J) < A (Max_Index'Result)))

function Max_Index (A : T_Arr) return Positive is
 Result : Positive := A'First;
begin
 for J in A'First .. A'Last loop
  if A (Result) < A (J) then
   Result := J;
  end if;
 end loop;
pragma Loop_Invariant
 ((for all K in A'First .. J => A (K) <= A (Result)) and
   (for all K in A'First .. Result - 1 =>
     A (K) < A (Result)));
return Result;
end Max_Index;
Ex: Inv holds initially

function Max_Index (A : T_Arr) return Positive is
  Result : Positive := A'First;
  J : A'First .. A'Last := A'First;
begin
  loop
    if A (Result) < A (J) then
      Result := J;
    end if;
  end loop;
  if J = A'Last then
    exit;
  else
    J := J + 1;
  end loop;
return Result;
end Max_Index;

Two paths to check
Ex: Inv holds initially

```vhdl
function Max_Index (A : T_Arr) return Positive is
  Result : Positive := A'First;
  J : A'First .. A'Last := A'First;
begin
  loop
    if A (Result) < A (J) then
      Result := J;
    end if;
  end loop;
  if J = A'Last then
    exit;
  else
    J := J + 1;
  end loop;
  return Result;
end Max_Index;
```

function Max_Index (A : T_Arr) return Positive is
  Result : Positive := A'First;
  J : A'First .. A'Last := A'First;
begin
  loop
    if A (Result) < A (J) then
      Result := J;
    end if;
    if J = A'Last then
      exit;
    else
      J := J + 1;
    end loop;
return Result;
end Max_Index;

(for all K in A'First .. J \to A(K) \leq A(J)) and
(for all K in A'First .. J - 1 \to A(K) < A(J))
function Max_Index (A : T_Arr) return Positive is
  Result : Positive := A'First;
  J : A'First .. A'Last := A'First;
begin
loop
  if A (Result) < A (J) then
    Result := J;
  end if;
  pragma Loop_Invariant ((for all K in A'First .. J => A (K) <= A (Result)) and (for all K in A'First .. J - 1 => A (K) < A (Result)));
  if J = A'Last then
    exit;
  else
    J := J + 1;
  end loop;
return Result;
end Max_Index;
function Max_Index (A : T_Arr) return Positive is
    Result : Positive := A'First;
    J : A'First .. A'Last := A'First;
begin
    loop
        if A (Result) < A (J) then
            Result := J;
        end if;
    end loop;
    return Result;
end Max_Index;
Example: Inv holds initially

function Max_Index (A : T_Arr) return Positive is
    Result : Positive := A'First;
    J : A'First .. A'Last := A'First;
begin
    loop
        if A (Result) < A (J) then
            Result := J;
        end if;
    end loop;
    return Result;
end Max_Index;
Example: Inv holds initially

function Max_Index (A : T_Arr) return Positive is
    Result : Positive := A'First;
    J : A'First .. A'Last := A'First;
begin
    loop
        if A (Result) < A (J) then
            Result := J;
        end if;
        if J = A'Last then
            exit;
        else
            J := J + 1;
        end loop;
    return Result;
end Max_Index;

A(A'First) < A(A'First)
→
(for all K in A'First .. A'First =>
    A(K) <= A(A'First)) and
(for all K in A'First .. A'First - 1 =>
    A(K) < A(A'First))

Trivially True b/c antecedent is False: Infeasible path!
Example: Inv holds initially

function Max_Index (A : T_Arr) return Positive is
  Result : Positive := A'First;
  J : A'First .. A'Last := A'First;
begin
  loop
    if A (Result) < A (J) then
      Result := J;
    end if;
  end loop;
  if J = A'Last then
    exit;
  else
    J := J + 1;
  end if;
return Result;
end Max_Index;

Pre: A'Length > 0

not A(A'First) < A(A'First)

→

(for all K in A'First .. A'First =>
  A(K) <= A(A'First)) and
(for all K in A'First .. A'First - 1 =>
  A(K) < A(A'First))

VC generated from this check guarantees invariant holds initially
Ex: Preservation of the loop invariant

```plaintext
function Max_Index (A : T_Arr) return Positive is
  Result : Positive := A'First;
  J : A'First .. A'Last := A'First;
begin
  loop
    if A (Result) < A (J) then
      Result := J;
    end if;
    pragma Loop_Invariant
      (for all K in A'First .. J => A (K) <= A (Result)) and
       (for all K in A'First .. Result - 1 =>
        A (K) < A (Result));
    if J = A'Last then
      exit;
    else
      J := J + 1;
    end loop;
  return Result;
end Max_Index;
```

Two paths to check b/c of the first conditional
Ex: Preservation of the loop invariant

function Max_Index (A : T_Arr) return Positive is
    Result : Positive := A'First;
    J : A'First .. A'Last := A'First;
begin
    loop
        if A (Result) < A (J) then
            Result := J;
        end if;
    pragma Loop_Invariant
        ((for all K in A'First .. J => A (K) <= A (Result)) and
         (for all K in A'First .. Result - 1 =>
            A (K) < A (Result)));
    if J = A'Last then
        exit;
    else
        J := J + 1;
    end loop;
    return Result;
end Max_Index;
Ex: Preservation of the loop invariant

function Max_Index (A : T_Arr) return Positive is
  Result : Positive := A'First;
  J : A'First .. A'Last := A'First;
begin
  loop
    if A (Result) < A (J) then
      Result := J;
    end if;
    pragma Loop_Invariant
      ((for all K in A'First .. J => A (K) <= A (Result)) and
       (for all K in A'First .. Result - 1 =>
        A (K) < A (Result)));
    if J = A'Last then
      exit;
    else
      J := J + 1;
    end loop;
  return Result;
end Max_Index;

not A(Result) < A(J)
→
(for all K in A'First .. J =>
  A(K) <= A(Result)) and
(for all K in A'First .. Result - 1 =>
  A(K) < A(Result))
Ex: Preservation of the loop invariant

```plaintext
function Max_Index (A : T_Arr) return Positive is
    Result : Positive := A'First;
    J : A'First .. A'Last := A'First;
begin
    loop
        if A (Result) < A (J) then
            Result := J;
        end if;
    pragma Loop_Invariant
        ((for all K in A'First .. J => A (K) <= A (Result)) and
        (for all K in A'First .. Result - 1 =>
            A (K) < A (Result)));
        if J = A'Last then
            exit;
        else
            J := J + 1;
        end if;
    end loop;
    return Result;
end Max_Index;
```

```
not A(Result) < A(J + 1)  
→  
(for all K in A'First .. J + 1 =>
    A(K) <= A(Result)) and
(for all K in A'First .. Result - 1 =>
    A(K) < A(Result))
```
function Max_Index (A : T_Arr) return Positive is
  Result : Positive := A'First;
  J : A'First .. A'Last := A'First;
begin
  loop
    if A (Result) < A (J) then
      Result := J;
    end if;

  pragma Loop_Invariant
    ((for all K in A'First .. J => A (K) <= A (Result)) and
     (for all K in A'First .. Result - 1 =>
      A (K) < A (Result)));

  if J = A'Last then
    exit;
  else
    J := J + 1;
  end loop;
  return Result;
end Max_Index;
function Max_Index (A : T_Arr) return Positive is
  Result : Positive := A'First;
  J : A'First .. A'Last := A'First;
begin
  loop
    if A (Result) < A (J) then
      Result := J;
    end if;
    pragma Loop_Invariant
      ((for all K in A'First .. J => A (K) <= A (Result)) and
       (for all K in A'First .. Result - 1 => A (K) < A (Result)));
    if J = A'Last then
      exit;
    else
      J := J + 1;
    end loop;
  return Result;
end Max_Index;

Assume inv:
  (for all K in A'First .. J => A(K) <= A(Result)) and
  (for all K in A'First .. Result - 1 => A(K) < A(Result))

not J = A'Last and
not A(Result) < A(J + 1) →
  (for all K in A'First .. J + 1 => A(K) <= A(Result)) and
  (for all K in A'First .. Result - 1 => A(K) < A(Result))
Ex: Preservation of the loop invariant

function Max_Index (A: T_Arr) return Positive is

Result := A'First;
J := A'First .. A'Last;
loop
if A (Result) < A (J) then
Result := J;
end if;
pragma Loop_Invariant ((for all K in A'First .. J => A (K) <= A (Result)) and
(for all K in A'First .. Result - 1 => A (K) < A (Result)));
if J = A'Last then
exit;
else
J := J + 1;
end loop;
return Result;
end Max_Index;

Assume inv:
(for all K in A'First .. J =>
A(K) <= A(Result)) and
(for all K in A'First .. Result - 1 =>
A(K) < A(Result))

not J = A'Last and
not A(Result) < A(J + 1)
→
(for all K in A'First .. J + 1 =>
A(K) <= A(Result)) and
(for all K in A'First .. Result - 1 =>
A(K) < A(Result))

VC generated from this check guarantees invariant is preserved on path that does not enter the 1st conditional
function Max_Index(A : T_Arr) return Positive is
  Result : Positive := A'First;
  J : A'First .. A'Last := A'First;
  begin
    loop
      if A (Result) < A (J) then
        Result := J;
      end if;
      pragma Loop_Invariant
      (for all K in A'First .. J => A (K) <= A (Result)) and
      (for all K in A'First .. Result - 1 => A (K) < A (Result));
    end loop;
    if J = A'Last then
      exit;
    else
      J := J + 1;
    end if;
  end loop;
  return Result;
end Max_Index;

Assume inv:
(for all K in A'First .. J => A(K) <= A(Result)) and
(for all K in A'First .. Result - 1 => A(K) < A(Result))

not J = A'Last
A(Result) < A(J + 1)
⇒
(for all K in A'First .. J + 1 => A(K) <= A(J + 1)) and
(for all K in A'First .. J + 1 - 1 => A(K) < A(J + 1))
Ex: Preservation of the loop invariant

function Max_Index
  Result : Positive
  J : A'First .. A'Last
begin
  loop
    if A (Result) < A (J) then
      Result := J;
    end if;
  pragma Loop_Invariant
    ((for all K in A'First .. J => A (K) <= A (Result)) and
     (for all K in A'First .. Result - 1 => A (K) < A (Result)));
  if J = A'Last then
    exit;
  else
    J := J + 1;
  end loop;
  return Result;
end Max_Index;

Assume inv:
(for all K in A'First .. J =>
  A(K) <= A(Result)) and
(for all K in A'First .. Result - 1 =>
  A(K) < A(Result))

VC generated from this check guarantees invariant is preserved on path that enters the 1st conditional
Ex: Post condition holds at termination

function Max_Index (A : T_Arr) return Positive is
  Result : Positive := A'First;
  J : A'First .. A'Last := A'First;
begin
  loop
    if A (Result) < A (J) then
      Result := J;
    end if;
  pragma Loop_Invariant
    ((for all K in A'First .. J => A (K) <= A (Result)) and
     (for all K in A'First .. Result - 1 =>
      A (K) < A (Result)));
    if J = A'Last then
      exit;
    else
      J := J + 1;
    end loop;
  return Result;
end Max_Index;

All paths to termination go through the loop: one path to check
Ex: Post condition holds at termination

function Max_Index (A : T_Arr) return Positive is
   Result : Positive := A'First;
   J : A'First .. A'Last := A'First;
begin
   loop
      if A (Result) < A (J) then
         Result := J;
      end if;
   pragma Loop_Invariant
      (for all K in A'First .. J => A (K) <= A (Result)) and
      (for all K in A'First .. Result - 1 =>
         A (K) < A (Result));
   if J = A'Last then
      exit;
   else
      J := J + 1;
   end loop;
   return Result;
end Max_Index;
Ex: Post condition holds at termination

function Max_Index (A : T_Arr) return Positive is
  Result : Positive := A'First;
  J : A'First .. A'Last := A'First;
begin
  loop
    if A (Result) < A (J) then
      Result := J;
    end if;
  pragma Loop_Invariant
    ((for all K in A'First .. J => A (K) <= A (Result)) and
     (for all K in A'First .. Result - 1 =>
      A (K) < A (Result)));
  if J = A'Last then
    exit;
  else
    J := J + 1;
  end loop;
return Result;
end Max_Index;

\[ WP(C, \text{"return exp"}) = C[FNAME\'Result/exp} \]

(for all K in A'Range =>
  A (K) <= A (Max_Index\'Result)) and
(for all K in A'First ..
  Max_Index\'Result - 1 =>
  A (K) < A (Max_Index\'Result))\]
Ex: Post condition holds at termination

function Max_Index (A : T_Arr) return Positive is
    Result : Positive := A'First;
    J : A'First .. A'Last := A'First;
begin
    loop
        if A (Result) < A (J) then
            Result := J;
        end if;
        pragma Loop_Invariant
        ((for all K in A'First .. J => A (K) <= A (Result)) and
         (for all K in A'First .. Result - 1 =>
          A (K) < A (Result)));
        if J = A'Last then
            exit;
        else
            J := J + 1;
        end loop;
    return Result;
end Max_Index;
function Max_Index

Result : Positive
J : A'First .. A'

begin
loop
if A (Result) < A (J) then
Result := J;
end if;
pragma Loop_Invariant
((for all K in A'First .. J => A (K) <= A (Result)) and
(for all K in A'First .. Result - 1 =>
A (K) < A (Result)));
if J = A'Last then
exit;
else
J := J + 1;
end loop;
return Result;
end Max_Index;

Assume inv:
(for all K in A'First .. J =>
A (K) <= A (Result)) and
(for all K in A'First .. Result - 1 =>
A (K) < A (Result))

J = A'Last

(for all K in A'Range =>
A (K) <= A (Result)) and
(for all K in A'First .. Result - 1 =>
A (K) < A (Result))
Ex: Post condition holds at termination

```vhd
function Max_Index (A : T_Arr) return Positive is
  Result : Positive := A'First;
  J : A'First .. A'Last := A'First;
  begin
    loop
      if A (Result) < A (J) then
        Result := J;
      end if;
      pragma Loop_Invariant (for all K in A'First .. J => A (K) <= A (Result) and (for all K in A'First .. Result - 1 => A (K) < A (Result)));
    end loop;
    if J = A'Last then
      exit;
    else
      J := J + 1;
    end loop;
    return Result;
  end Max_Index;
```

Assume inv:
- (for all K in A'First .. J => A(K) <= A(Result)) and (for all K in A'First .. Result - 1 => A(K) < A(Result))
- J = A'Last
  - (for all K in A'Range => A (K) <= A (Result)) and (for all K in A'First .. Result - 1 => A (K) < A (Result))

VC generated from this check guarantees post condition is true at termination
Ex: Post condition holds at termination

function Max_Index (A : T_Arr) return Positive is
  Result : Positive := A'First;
  J : A'First .. A'Last := A'First;
begin
  loop
    if A (Result) < A (J) then
      Result := J;
    end if;
    pragma Loop_Invariant
      ((for all K in A'First .. J => A (K) <= A (Result)) and
       (for all K in A'First .. Result - 1 =>
        A (K) < A (Result)));
    if J = A'Last then
      exit;
    else
      J := J + 1;
    end loop;
  return Result;
end Max_Index;
Acknowledgements & references
