Proof of Functional Correctness

Presented by Claire Dross and Martyn Pike

University.adacore.com

Functional Correctness – Beyond Program Integrity

- **Proof of functional correctness verifies that a program complies with its specification**
  - Verifies additional properties that can be derived from requirements, documentation, comments, or from test results
  - These additional properties may not be necessary for program integrity (verifying that no error can occur at runtime)
- **To be verified by GNATprove, the specification should be expressed as SPARK contracts**

```
function Find (A : Nat_Array; E : Natural) return Natural with
  Post => Find'Result in 0 | A'Range;

function Find (A : Nat_Array; E : Natural) return Natural with
  Post =>
   (if (for all I in A'Range => A (I) /= E)
    then Find'Result = 0
    else Find'Result in A'Range and then A (Find'Result) = E);
```

Functional Correctness – Beyond Program Integrity

- **Contracts for functional correctness can be executed**
  - Executable specifications can stand for test oracles
  - If they are checked during integration testing, they may replace unit testing
  - They are more maintainable than comments
- **… as well as formally proven**
  - Formal proof verifies the program on all possible inputs
  - Verification can be done earlier in the development, before bodies are implemented

```
function Find (A : Nat_Array; E : Natural) return Natural
with Post => ...

procedure Use_Find is
...
  Find (A, E);
...
```

Functional Correctness – Advanced Contracts

- **Ada 2012 constructs allow to express powerful contracts for functional correctness**
  - Syntax of expressions are extended with quantified and case expressions
  - Expression functions can be used to factorize them for readability
  - To express an invariant specific to an object, type predicates can be more adapted than subprogram contracts

```
function Is_Sorted (A : Nat_Array) return Boolean is
  (for all I in A'Range =>
    (if I < A'Last then A (I) <= A (I + 1)));
--  Returns True if A is sorted in increasing order.

subtype Sorted_Nat_Array is Nat_Array with
  Dynamic_Predicate => Is_Sorted (Sorted_Nat_Array);
--  Elements of type Sorted_Nat_Array are all sorted.
```

Functional Correctness – Advanced Contracts – Ghost Code

- **Ghost code is normal Ada code that is only used for specification**
  - Ghost code should have no effect on the behavior of the program
  - When the program is compiled with assertions, ghost code is executed like normal code
  - The compiler can also be instructed not to generate code for it
- **Ghost entities should be marked with the Ghost aspect**

```
procedure Do_Something (X : in out T) is
   X_Init : constant T := X with Ghost;
begin
   Do_Some_Complex_Stuff (X);
   pragma Assert (Is_Correct (X_Init, X));
   --  It is OK to use X_Init inside an assertion.

   X := X_Init;
   --  Compilation error:
   --     Ghost entity cannot appear in this context.
```

Functional Correctness – Advanced Contracts – Ghost Functions

- **Ghost functions express properties only used in contracts**
  - Expression functions used to factor out common expression in contracts can be marked as ghost
  - Ghost functions can also be used to disclose state abstractions for specification purpose
  - Ghost functions can be inefficient as long as assertions are disabled in the final executable

```
type Stack is private;

function Get_Model (S : Stack) return Nat_Array with Ghost;
--  Returns an array as a model of a stack.

procedure Push (S : in out Stack; E : Natural) with
  Pre  => Get_Model (S)'Length < Max,
  Post => Get_Model (S) = Get_Model (S)'Old & E;

function Peek (S : Stack; I : Positive) return Natural is
  (Get_Model (S) (I));
-- Get_Model cannot be used in this context.
```

Functional Correctness – Advanced Contracts – Global Ghost Variables

- **Global ghost variables store information that is only useful for specification**
  - They can be used to maintain a model of a complex or private data-structure
  - To specify properties over several runs of subprograms
  - Or to act as placeholders for intermediate values of variables

```
Last_Accessed_Is_A : Boolean := False with Ghost;
procedure Access_A with
  Post => Last_Accessed_Is_A;
procedure Access_B with
  Pre  => Last_Accessed_Is_A,
  Post => not Last_Accessed_Is_A;
  --  B can only be accessed after A

V_Interm : T with Ghost;
procedure Do_Two_Things (V : in out T) with
  Post => (First_Thing_Done (V'Old, V_Interm)
       and Second_Thing_Done (V_Interm, V));
```

Functional Correctness – Guide Proof

- **As contracts for functional correctness can be complex, users may need to guide the proof tool**
  - Intermediate assertions may help the tool verify a complex reasoning
  - It may be useful to express the property we want to verify in a different way, even if it is theoretically equivalent
  - Remaining unproved assertions can be discharged by test or by review

```
pragma Assert (Assertion_Checked_By_The_Tool);
--  info: assertion proved
pragma Assert (Assumption_Validated_By_Other_Means);
--  medium: assertion might fail

pragma Assume (Assumption_Validated_By_Other_Means);
--  The tool does not attempt to check this expression.
--  It is recorded as an assumption.
```

Functional Correctness – Guide Proof – Local Ghost Variables

- **Local ghost variables or constants store information that is only useful in intermediate assertions**
  - They can be used to store the intermediate values of variables or expressions
  - Or to construct iteratively a data structure that reflects a property of the subprogram

```
procedure P (X : in out T) with Post => F (X, X'Old) is
  X_Init : constant T := X with Ghost;
begin
  if Condition (X) then
    ...
    pragma Assert (F (X, X_Init));
...

procedure Sort (A : in out Nat_Array) with
  Post => (for all I in A'Range =>
             (for some J in A'Range => A (I) = A'Old (J))) is
  Permutation : Index_Array := (1 => 1, 2 => 2, ...) with
    Ghost;
```

Functional Correctness – Guide Proof – Ghost Procedures

- **Ghost procedures can only affect the values of Ghost variables**
  - They can be used to abstract away complex treatment on ghost variables
    - In normal code, the only statements that can refer to ghost entities are assignments to ghost variables and ghost procedure calls
  - Or to group together intermediate assertions

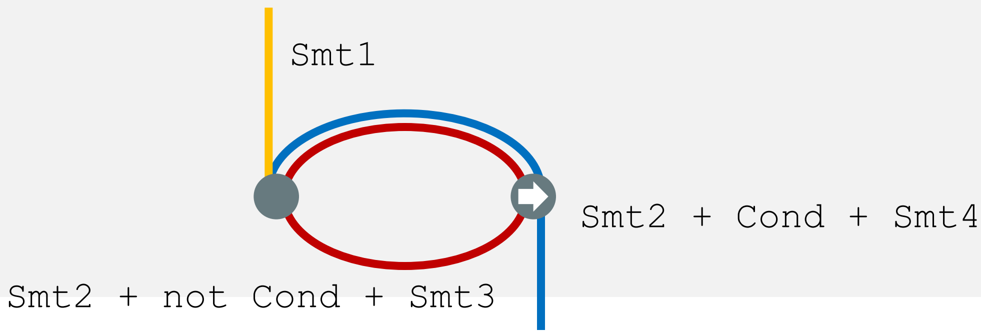
```
A : Nat_Array := ... with Ghost;
procedure Increase_A with Ghost is
begin
  for I in A'Range loop
    A (I) := A (I) + 1;
  end loop;
end Increase_A;

procedure Prove_P (X : T) with Ghost,
  Global => null,
  Post   => P (X);
```

Functional Correctness – Guide Proof – Handling of Loops

- **GNATprove handles subprograms with loops by dividing them into several parts, so as to flatten the control flow**
  - From the subprogram's beginning to the loop
  - From the loop's beginning to the loop's end (one loop iteration)
    - The exit conditions are assumed not to hold
    - The initial values of variables modified in the loop are unknown
  - From the loop's beginning to subprogram's end for each exit path
    - The exit conditions are assumed to hold
    - The initial values of variables modified in the loop are unknown

```
Stmt1;
loop
  Stmt2;
  exit when Cond;
  Stmt3;
end loop;
Stmt4;
```



Functional Correctness – Guide Proof – Handling of Loops

- **Information about modifications of variables by previous iterations of the loop is lost**
  - The tool knows only about values of unmodified variables
  - And about the loop's range or condition if any
- **No information about constants is accumulated**

```
function Find (A : Nat_Array; E : Natural) return Natural is
begin
  for I in A'Range loop
    pragma Assert
          (for all J in A'First .. I - 1 => A (J) /= E);
    --  medium: assertion might fail
    if A (I) = E then
      return I;
    end if;
    pragma Assert (A (I) /= E);
    --  info: assertion proved
  end loop;
  return 0;
end Find;
```

Functional Correctness – Guide Proof – Loop Invariants

- **Loop invariants are assertions which should be true at every iteration of the loop**
  - They are usually written at the beginning of the loop but can be placed anywhere else provided it is at the top-level
  - GNATprove performs two checks to ensure their validity:
    - Loop invariant initialization: the property holds in the first iteration
    - Loop invariant preservation: the property holds in an arbitrary iteration of the loop, provided it holds in the previous one

```
function Find (A : Nat_Array; E : Natural) return Natural is
begin
  for I in A'Range loop
    pragma Loop_Invariant (A (A'First) /= E);
    --  medium: loop invariant might fail in first iteration
    --  info: loop invariant preservation proved
    if A (I) = E then
      return I;
    end if;
  end loop;
```

Functional Correctness – Guide Proof – Loop Invariants

- **Loop invariants are used as cut points to verify loops**
  - The loop invariant initialization is checked in the first iteration of the loop
  - The body of the loop, the loop invariant preservation, and the program statements following the loop are all checked assuming that the loop invariant holds in the previous iteration.

```
function Find (A : Nat_Array; E : Natural) return Natural is
begin
  for I in A'Range loop
    pragma Loop_Invariant
          (for all J in A'First .. I - 1 => A (J) /= E);
    --  info: loop invariant initialization proved
    --  info: loop invariant preservation proved
    if A (I) = E then
      return I;
    end if;
  end loop;
  pragma Assert (for all I in A'Range => A (I) /= E);
  --  info: assertion proved
```

Functional Correctness – Guide Proof – Loop Invariants

- **In practice, an invariant should have 4 good properties:**
  - [INIT] It should be provable in the first iteration of the loop
  - [INSIDE] It should allow proving absence of run-time errors and local assertions inside the loop
  - [AFTER] It should allow proving absence of run-time errors, local assertions and the subprogram postcondition after the loop
  - [PRESERVE] It should be provable after the first iteration of the loop

```
A_I : constant Nat_Array := A with Ghost;
for K in A'Range loop
 A (K) := F (A (K));
 pragma Loop_Invariant
   (for all J in A'First .. K => A (J) = F (A'Loop_Entry (J)));
  --  info: loop invariant initialization proved
  --  info: loop invariant preservation proved
end loop;
pragma Assert (for all K in A'Range => A (K) = F (A_I (K)));
--  info: assertion proved
```

Quiz

Is this correct? 1/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
package Ring_Buffer with SPARK_Mode is
  function Valid_Model (M : Nat_Array) return Boolean;

  function Get_Model return Nat_Array with Ghost,
    Post => Valid_Model (Get_Model'Result)
      and Get_Model'Result'First = 1
      and Get_Model'Result'Length in Length_Range;

  procedure Push_Last (E : Natural) with
    Pre  => Get_Model'Length < Max_Size,
    Post => Get_Model = Get_Model'Old & E;

  ...
end Ring_Buffer;

package body Ring_Buffer with SPARK_Mode is
  Content : Nat_Array (1 .. Max_Size);
  First   : Index_Range;
  Length  : Length_Range;

  function Get_Model return Nat_Array is
    Result : Nat_Array (1 .. Length);
  begin
    ...
```

Is this correct? 1/10

YES

```
package Ring_Buffer with SPARK_Mode is
  function Valid_Model (M : Nat_Array) return Boolean;

  function Get_Model return Nat_Array with Ghost,
    Post => Valid_Model (Get_Model'Result)
    and Get_Model'Result'First = 1
    and Get_Model'Result'Length in Length_Range;

  procedure Push_Last (E : Natural) with
    Pre  => Get_Model'Length < Max_Size,
    Post => Get_Model = Get_Model'Old & E;

  ...
end Ring_Buffer;

package body Ring_Buffer with SPARK_Mode is
  Content : Nat_Array (1 .. Max_Size);
  First   : Index_Range;
  Length  : Length_Range;

  function Get_Model return Nat_Array is
    Result : Nat_Array (1 .. Length);
  begin
```

This is correct as Get_Model is used for specification only. Note that calls to Get_Model cause copies of the buffer's content. They can be automatically removed from production code by the compiler.

Is this correct? 2/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
package Ring_Buffer with SPARK_Mode is
   type Model_Type (Length : Length_Range := 0) is record
     Content : Nat_Array (1 .. Length);
   end record with Ghost;
   Model : Model_Type with Ghost;

   function Valid_Model return Boolean;

   procedure Push_Last (E : Natural) with
     Pre  => Valid_Model and Model.Length < Max_Size,
     Post => Valid_Model and Model.Content = Model.Content'Old & E;

   ...
end Ring_Buffer;

package body Ring_Buffer with SPARK_Mode is
   Content : Nat_Array (1 .. Max_Size);
   First   : Index_Range;
   Length  : Length_Range;

   function Valid_Model return Boolean is
     (Length = Model.Length and then ...);

   ...
end Ring_Buffer;
```

Is this correct? 2/10

NO

```
package Ring_Buffer with SPARK_Mode is
  type Model_Type (Length : Length_Range := 0) is record
    Content : Nat_Array (1 .. Length);
  end record with Ghost;
  Model : Model_Type with Ghost;

  function Valid_Model return Boolean;

  procedure Push_Last (E : Natural) with
    Pre  => Valid_Model and Model.Length < Max_Size,
    Post => Valid_Model and Model.Content = Model.Content'Old & E;

  ...
end Ring_Buffer;

package body Ring_Buffer with SPARK_Mode is
  Content : Nat_Array (1 .. Max_Size);
  First   : Index_Range;
  Length  : Length_Range;

  function Valid_Model return Boolean is
    (Length = Model.Length and then ...);
```

Model, which is a ghost variable, cannot influence the return value of the normal function Valid_Model. As Valid_Model is only used in specifications, it could be marked as ghost.

Is this correct? 3/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
Model : Model_Type with Ghost;

procedure Pop_When_Available (E : in out Natural) with
  Pre              => Valid_Model,
  Post             => Valid_Model,
  Contract_Cases =>
     (Model.Length > 0 => E & Model.Content = Model.Content'Old,
      others           => Model = Model’Old and E = E’Old);

procedure Pop_When_Available (E : in out Natural) is
begin
  if Length > 0 then
    Model := (Length  => Model.Length - 1,
              Content => Model.Content (2 .. Model.Length));
    E := Content (First);
    Length := Length - 1;
    First := (if First < Max_Size then First + 1 else 1);
  end if;
end Pop_When_Available;
```

Is this correct? 3/10

YES

```
Model : Model_Type with Ghost;

procedure Pop_When_Available (E : in out Natural) with
  Pre            => Valid_Model,
  Post           => Valid_Model,
  Contract_Cases =>
    (Model.Length > 0 => E & Model.Content = Model.Content'Old,
     others           => Model = Model'Old and E = E'Old);

procedure Pop_When_Available (E : in out Natural) is
begin
  if Length > 0 then
    Model := (Length  => Model.Length - 1,
              Content => Model.Content (2 .. Model.Length));
    E := Content (First);
    Length := Length - 1;
    First := (if First < Max_Size then First + 1 else 1);
  end if;
end Pop_When_Available;
```

Model, though it is marked as Ghost, can be referenced from the body of the nonghost procedure Pop_When_Available as long as it is only used in ghost statements.

Is this correct? 4/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
Model : Model_Type with Ghost;

procedure Pop_When_Available (E : in out Natural) with
  Pre            => Valid_Model,
  Post           => Valid_Model,
  Contract_Cases =>
    (Model.Length > 0 => E & Model.Content = Model.Content'Old,
     others           => Model = Model'Old and E = E'Old);

procedure Pop_When_Available (E : in out Natural) is
begin
  if Model.Length > 0 then
    Model := (Length  => Model.Length - 1,
              Content => Model.Content (2 .. Model.Length));
  end if;

  if Length > 0 then
    E := Content (First);
    Length := Length - 1;
    First := (if First < Max_Size then First + 1 else 1);
  end if;
end Pop_When_Available;
```

Is this correct? 4/10

NO

```
Model : Model_Type with Ghost;

procedure Pop_When_Available (E : in out Natural) with
  Pre           => Valid_Model,
  Post          => Valid_Model,
  Contract_Cases =>
    (Model.Length > 0 => E & Model.Content = Model.Content'Old,
     others           => Model = Model’Old and E = E’Old);

procedure Pop_When_Available (E : in out Natural) is
begin
  if Model.Length > 0 then
    Model := (Length  => Model.Length - 1,
              Content => Model.Content (2 .. Model.Length));
  end if;

  if Length > 0 then
    E := Content (First);
    Length := Length - 1;
    First := (if First < Max_Size then First + 1 else 1);
  end if;
end Pop_When_Available;
```

The test on Model is not allowed even though it is only used to update its own value. Indeed, to simplify removal of ghost code by the compiler, the only statements considered as ghost in normal code are assignments to ghost variables and ghost procedure calls.

Is this correct? 5/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
Model : Model_Type with Ghost;

procedure Pop_When_Available (E : in out Natural) is
  procedure Update_Model with Ghost is
  begin
    if Model.Length > 0 then
      Model := (Length  => Model.Length - 1,
                Content => Model.Content (2 .. Model.Length));
    end if;
  end Update_Model;

begin
  Update_Model;

  if Length > 0 then
    E := Content (First);
    Length := Length - 1;
    First := (if First < Max_Size then First + 1 else 1);
  end if;
end Pop_When_Available;
```

Is this correct? 5/10

YES

```
Model : Model_Type with Ghost;

procedure Pop_When_Available (E : in out Natural) is
  procedure Update_Model with Ghost is
  begin
    if Model.Length > 0 then
      Model := (Length  => Model.Length - 1,
                Content => Model.Content (2 .. Model.Length));
    end if;
  end Update_Model;

begin
  Update_Model;

  if Length > 0 then
    E := Content (First);
    Length := Length - 1;
    First := (if First < Max_Size then First + 1 else 1);
  end if;
end Pop_When_Available;
```

Everything is fine here. Model is only accessed inside Update_Model which is itself a ghost procedure. Moreover, we don't need to add any contract to Update_Model. Indeed, as it is a local procedure, it will be inlined by GNATprove.

Is this correct? 6/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
function Max_Array (A, B : Nat_Array) return Nat_Array with
  Pre => A'Length = B'Length;

function Max_Array (A, B : Nat_Array) return Nat_Array is
  R : Nat_Array (A'Range) := (others => 0);
  J : Integer := B'First;
begin
  for I in A'Range loop
    if A (I) > B (J) then
      R (I) := A (I);
    else
      R (I) := B (J);
    end if;
    J := J + 1;
  end loop;
  return R;
end Max_Array;
```

Is this correct? 6/10

YES

```
function Max_Array (A, B : Nat_Array) return Nat_Array with
  Pre => A'Length = B'Length;

function Max_Array (A, B : Nat_Array) return Nat_Array is
  R : Nat_Array (A'Range) := (others => 0);
  J : Integer := B'First;
begin
  for I in A'Range loop
    if A (I) > B (J) then
      R (I) := A (I);
    else
      R (I) := B (J);
    end if;
    J := J + 1;
  end loop;
  return R;
end Max_Array;
```

This program is correct. Unfortunately, GNATprove will fail to verify that J stays in the index range of B. Indeed, when checking the body of the loop, GNATprove forgets everything about the current value of J as it will have been modified by previous iterations of the loop. To get more precise results, we need to provide a loop invariant.

Is this correct? 7/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
function Max_Array (A, B : Nat_Array) return Nat_Array with
  Pre => A'Length = B'Length;

function Max_Array (A, B : Nat_Array) return Nat_Array is
  R : Nat_Array (A'Range) := (others => 0);
  J : Integer := B'First;
begin
  for I in A'Range loop
    pragma Loop_Invariant (J in B'Range);
    if A (I) > B (J) then
      R (I) := A (I);
    else
      R (I) := B (J);
    end if;
    J := J + 1;
  end loop;
  return R;
end Max_Array;
```

Is this correct? 7/10

YES

```
function Max_Array (A, B : Nat_Array) return Nat_Array with
  Pre => A'Length = B'Length;

function Max_Array (A, B : Nat_Array) return Nat_Array is
  R : Nat_Array (A'Range) := (others => 0);
  J : Integer := B'First;
begin
  for I in A'Range loop
    pragma Loop_Invariant (J in B'Range);
    if A (I) > B (J) then
      R (I) := A (I);
    else
      R (I) := B (J);
    end if;
    J := J + 1;
  end loop;
  return R;
end Max_Array;
```

The loop invariant now allows verifying that no runtime error can occur in the loop's body. Unfortunately, GNATprove will fail to verify that the invariant stays valid after the first iteration of the loop. Indeed, knowing that J is in B'Range in a given iteration is not enough to show that it will remain so in the next iteration. We need a more precise invariant, linking J to the value of the loop index I, like J = I – A'First + B'First.

Is this correct? 8/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
function Max_Array (A, B : Nat_Array) return Nat_Array with
  Pre  => A'First = B'First and A'Last = B'Last,
  Post =>
   (for all K in A'Range =>
         Max_Array'Result (K) = Natural'Max (A (K), B (K)));

function Max_Array (A, B : Nat_Array) return Nat_Array is
  R : Nat_Array (A'Range) := (others => 0);
begin
  for I in A'Range loop
    pragma Loop_Invariant (for all K in A'First .. I =>
                 R (K) = Natural'Max (A (K), B (K)));
    if A (I) > B (I) then
      R (I) := A (I);
    else
      R (I) := B (I);
    end if;
  end loop;
  return R;
end Max_Array;
```

Is this correct? 8/10

NO

```
function Max_Array (A, B : Nat_Array) return Nat_Array with
  Pre  => A'First = B'First and A'Last = B'Last,
  Post =>
   (for all K in A'Range =>
         Max_Array'Result (K) = Natural'Max (A (K), B (K)));

function Max_Array (A, B : Nat_Array) return Nat_Array is
  R : Nat_Array (A'Range) := (others => 0);
begin
  for I in A'Range loop
    pragma Loop_Invariant (for all K in A'First .. I =>
               R (K) = Natural'Max (A (K), B (K)));
    if A (I) > B (I) then
      R (I) := A (I);
    else
      R (I) := B (I);
    end if;
  end loop;
  return R;
end Max_Array;
```

The program itself is correct but the invariant is not, as can be checked by executing the function Max_Array with assertions enabled. Indeed, at each loop iteration, R contains the maximum of A and B only until I - 1 as the Ith Index was not handled yet.

Is this correct? 9/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
procedure Max_Array (A : in out Nat_Array, B : Nat_Array) with
  Pre  => A'First = B'First and A'Last = B'Last,
  Post => (for all K in A'Range =>
              A (K) = Natural'Max (A'Old (K), B (K)));

procedure Max_Array (A : in out Nat_Array, B : Nat_Array) is
begin
  for I in A'Range loop
    pragma Loop_Invariant (for all K in A'First .. I - 1 =>
              A (K) = Natural'Max (A'Loop_Entry (K), B (K)));
    pragma Loop_Invariant
      (for all K in I .. A'Last => A (K) = A'Loop_Entry (K));
    if A (I) <= B (I) then
      A (I) := B (I);
    end if;
  end loop;
end Max_Array;
```

Is this correct? 9/10

YES

```
procedure Max_Array (A : in out Nat_Array, B : Nat_Array) with
  Pre  => A'First = B'First and A'Last = B'Last,
  Post => (for all K in A'Range =>
              A (K) = Natural'Max (A'Old (K), B (K)));

procedure Max_Array (A : in out Nat_Array, B : Nat_Array) is
begin
  for I in A'Range loop
    pragma Loop_Invariant (for all K in A'First .. I - 1 =>
               A (K) = Natural'Max (A'Loop_Entry (K), B (K)));
    pragma Loop_Invariant
      (for all K in I .. A'Last => A (K) = A'Loop_Entry (K));
    if A (I) <= B (I) then
      A (I) := B (I);
    end if;
  end loop;
end Max_Array;
```

The program is correct. GNATprove can verify that the loop invariant stays valid after the first iteration thanks to the provided frame condition: it knows that the values stored in A after I were preserved in the previous iterations.

Is this correct? 10/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
procedure Max_Array (A : in out Nat_Array, B : Nat_Array) with
  Pre  => A'First = B'First and A'Last = B'Last,
  Post => (for all K in A'Range =>
               A (K) = Natural'Max (A'Old (K), B (K)));

procedure Max_Array (A : in out Nat_Array, B : Nat_Array) is
begin
  for I in A'Range loop
    pragma Loop_Invariant (for all K in A'First .. I - 1 =>
                A (K) = Natural'Max (A'Loop_Entry (K), B (K)));
    if A (I) <= B (I) then
      A (I) := B (I);
    end if;
  end loop;
end Max_Array;
```

Is this correct? 10/10

YES

```
procedure Max_Array (A : in out Nat_Array, B : Nat_Array) with
  Pre  => A'First = B'First and A'Last = B'Last,
  Post => (for all K in A'Range =>
              A (K) = Natural'Max (A'Old (K), B (K)));

procedure Max_Array (A : in out Nat_Array, B : Nat_Array) is
begin
  for I in A'Range loop
    pragma Loop_Invariant (for all K in A'First .. I - 1 =>
                A (K) = Natural'Max (A'Loop_Entry (K), B (K)));
    if A (I) <= B (I) then
      A (I) := B (I);
    end if;
  end loop;
end Max_Array;
```

The program is correct. GNATprove can verify that the loop invariant stays valid after the first iteration thanks to its generation of the frame condition: it knows that the values stored in A after I were preserved in the previous iterations.

university.adacore.com