SPARK 2014: Proof of Program Integrity

Claire Dross and Martyn Pike

University.adacore.com

Proof of Program Integrity – Run-Time Errors

- **A runtime error is an error occurring at program execution (as opposed to at program compilation)**
  - Among them are array access errors, range violations, overflows in computations…

```
type Nat_Array is array (Integer range <>) of Natural;

A : Nat_Array (1 .. 10);
I, J, P, Q : Integer;

A (I + J) := P / Q;
```

- **Several runtime errors may occur when executing this program**

Proof of Program Integrity – Run-Time Errors

```
A (I + J) := P / Q;
```

- **The following errors could occur:**
  - I+J might overflow

```
A (Integer'Last + 1) := P / Q;
```

  - I+J might be outside of the A's index range

```
A (A'Last + 1) := P / Q;
```

  - P/Q might overflow

```
A (I + J) := Integer'First / -1;
```

  - P/Q might be outside the element subtype

```
A (I + J) := 1 / -1;
```

  - Q might be zero

```
A (I + J) := P / 0;
```

Proof of Program Integrity – Run-Time Errors

- **By default, the compiler will generate an executable that contains checks**
  - So that an exception is raised if a runtime error occurs
  - For some runtime errors, an additional switch is required at compilation to get these checks

```
A (Integer'Last + 1) := P / Q;
raised CONSTRAINT_ERROR : overflow check failed

A (A'Last + 1) := P / Q;
raised CONSTRAINT_ERROR : index check failed

A (I + J) := Integer'First / (-1);
raised CONSTRAINT_ERROR : overflow check failed

A (I + J) := 1 / (-1);
raised CONSTRAINT_ERROR : range check failed

A (I + J) := P / 0;
raised CONSTRAINT_ERROR : divide by zero
```

Proof of Program Integrity – Run-Time Errors

- **GNATprove can statically verify absence of run-time errors**
  - A logical formula, named Verification Condition, is generated for each possible runtime error
  - If the Verification Condition is true, the error cannot occur

```
A (Integer'Last + 1) := P / Q;
medium: overflow check might fail

A (A'Last + 1) := P / Q;
medium: array index check might fail

A (I + J) := Integer'First / (-1);
medium: overflow check might fail

A (I + J) := 1 / (-1);
medium: range check might fail

A (I + J) := P / 0;
medium: divide by zero might fail
```

Proof of Program Integrity – Modularity

- **GNATprove uses contracts to perform proof of program modularly on a per subprogram basis**
  - When verifying a subprogram’s body, its precondition is all that is known about its inputs
  - When a subprogram is called, its post-condition is all that is known about its outputs

```
procedure Increment (X : in out Integer) with
   Pre => X < Integer'Last is
begin
   X := X + 1;
-- info: overflow check proved
end;

X := Integer'Last - 2;
Increment (X);
-- Here GNATprove does not know the value of X
X := X + 1;
-- medium: overflow check might fail
```

Proof of Program Integrity – Modularity – Exceptions

- **Local subprograms without contracts can be inlined**

```
procedure Increment (X : in out Integer) is
begin
   X := X + 1;
-- info: overflow check proved, in call inlined at line 7
end Increment;

X := Integer'Last - 2;
Increment (X);
X := X + 1;
-- info: overflow check proved
```

- **Values of expression functions are known**

```
function Increment (X : Integer) return Integer is
    (X + 1)
-- info: overflow check proved
with Pre => X < Integer'Last;

X := Integer'Last - 2;
X := Increment (X);
X := X + 1;
-- info: overflow check proved
```

Proof of Program Integrity – Contracts

- **Ada 2012 contracts can be verified at runtime**
  - Runtime checks can be introduced for them during compilation
  - If they do not hold at a subprogram call, an exception is raised

```
procedure Increment (X : in out Integer) with
  Pre => X < Integer’Last;
X := Integer’Last;
Increment (X);
-- raised ASSERT_FAILURE : failed precondition

procedure Absolute (X : in out Integer) with
  Post => X >= 0 is
begin
  if X > 0 then
    X := - X;
  end if;
end Absolute;
X := 1;
Absolute (X);
-- raised ASSERT_FAILURE : failed postcondition
```

Proof of Program Integrity – Contracts

- **GNATprove statically verifies pre and postconditions**
  - Preconditions are checked at each subprogram call
  - Postconditions are verified once as part of the subprogram's body

```
procedure Increment (X : in out Integer) with
  Pre => X < Integer'Last;
X := Integer'Last;
Increment (X);
-- medium: precondition might fail

procedure Absolute (X : in out Integer) with
  Post => X >= 0 is
-- medium: postcondition might fail, requires X >= 0
begin
  if X > 0 then
    X := - X;
  end if;
end Absolute;
X := 1;
Absolute (X);
```

Proof of Program Integrity – Contracts – Executable Semantics

- **The semantics in contracts are the usual Ada semantics**
  - Facilitate combination with testing on a per subprogram basis
  - Facilitate debugging of assertions, as they can be executed
- **GNATprove also verifies integrity of assertions**
  - Proof of absence or runtime errors in preconditions is done once as part of the verification of the subprogram's body
  - A switch can be used to disable overflow checks in assertions both for execution and static verification

```
function Add (X, Y : Integer) return Integer with
  Pre => X + Y in Integer;
-- medium: overflow check might fail

X := Add (Integer'Last, 1);
-- raised CONSTRAINT_ERROR : overflow check failed
```

Proof of Program Integrity – Contracts – Additional Contracts

- **Assume generates an assumption for the proof tool**
- **Contract-Cases allows to annotate a subprogram by specifying a disjoint set of cases**
  - A case is composed of a guard and a consequence
  - Guards of different cases must be disjoint and complete (there is always one and only one active case).
  - Only the consequence of the active case is verified

```
procedure Absolute (X : in out Integer) with
  Pre            => X > Integer'First,
  Contract_Cases => (X < 0  => X = - X'Old,
                     X >= 0 => X = X'Old);
-- info: disjoint contract cases proved
-- info: complete contract cases proved
-- info: contract case proved
pragma Assume (X < Integer'Last);
X := X + 1;
```

Proof of Program Integrity – Debug Failed Proof Attempts (1)

- **Investigating Incorrect Code or Assertion**
  - [CODE] The check or assertion does not hold, because the code is wrong.
  - [ASSERT] The assertion does not hold, because it is incorrect.

```
procedure Incr_Until (X : in out Natural) with
  Contract_Cases =>
     (Incremented => X > X'Old,
-- medium: contract case might fail
      others      => X = X'Old) is
-- medium: contract case might fail
begin
  if X < 1000 then
    X := X + 1;
    Incremented := True;
  else
    Incremented := False;
  end if;
end Incr_Until;
```

Proof of Program Integrity – Debug Failed Proof Attempts (2)

- **Investigating Incorrect Code or Assertion**
  - Test the program on representative inputs with the assertions enabled

```
procedure Incr_Until (X : in out Natural) with
  Contract_Cases =>
    (Incremented => X > X'Old,
     others      => X = X'Old) is
begin
  …
end Incr_Until;

X := 0;
Incr_Until (X);

X := 1000;
Incr_Until (X);
-- raised ASSERT_FAILURE : failed contract case at line 3

-- Incremented is True when evaluating the
-- Contract_Cases' guards?
-- That is because they are evaluated before the call!
```

Proof of Program Integrity – Debug Failed Proof Attempts (3)

- **Investigating Unprovable Properties**
  - [SPEC] The check or assertion cannot be proved, because of some missing assertions about the behavior of the program
  - [MODEL] The check or assertion is not proved because of current limitations in the model used by GNATprove

```
C : Natural := 100;
procedure Increase (X : in out Natural) with
  Post => (if X < C then X > X'Old else X = C) is
-- medium: postcondition might fail
begin
  if X < 90 then
    X := X + 10;
  elsif X >= C then
    X := C;
  else
    X := X + 1;
  end if;
end Increase;
```

Proof of Program Integrity – Debug Failed Proof Attempts (4)

- **Investigating Unprovable Properties**
  - GNATprove can precisely locate the part of large assertions that it cannot prove
  - It also provides path information that might help the code review
  - It may be useful to simplify the code during this investigation, for example by adding a simpler assertion and trying to prove it

```
C : Natural := 100; -- Requires C >= 90

procedure Increase (X : in out Natural) with
   Post => (if X < C then X > X'Old else X = C) is
-- medium: postcondition might fail, requires X = C
begin
   if X < 90 then
      X := X + 10;
   elsif X >= C then
      X := C;
…
```

Proof of Program Integrity – Debug Failed Proof Attempts (5)

- **Investigating Prover Shortcomings**
  - [TIMEOUT] The check or assertion is not proved because the prover timeouts
  - [PROVER] The check or assertion is not proved because the prover is not smart enough

```
function GCD (A, B : Positive) return Positive with
  Post => A mod GCD'Result = 0
        and B mod GCD'Result = 0 is
-- medium: postcondition might fail
begin
  if A > B then
    return GCD (A - B, B);
  elsif B > A then
    return GCD (A, B - A);
  else
    return A;
  end if;
end GCD;
```

Proof of Program Integrity – Debug Failed Proof Attempts (6)

- **Investigating Prover Shortcomings**
  - Test if the prover would find a proof given more time or if another prover can find a proof
  - Non-linear arithmetic (multiplication, division, modulo…) is not well handled by provers in general

```
function GCD (A, B : Positive) return Positive with
  Post => A mod GCD'Result = 0
        and B mod GCD'Result = 0 is
begin
  if A > B then
    Result := GCD (A - B, B);
    pragma Assert ((A - B) mod Result = 0);
-- info: assertion proved
    pragma Assert (B mod Result = 0);
-- info: assertion proved
    pragma Assert (A mod Result = 0);
-- medium: assertion might fail
```

Quiz

Is this correct? 1/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
package Lists with SPARK_Mode is
  function Goes_To (I, J : Index) return Boolean;

  procedure Link (I, J : Index) with Post => Goes_To (I, J);
private
  type Cell (Is_Set : Boolean := True) is record …
  type Cell_Array is array (Index) of Cell;

  Memory : Cell_Array;
end Lists;

package body Lists with SPARK_Mode is
   function Goes_To (I, J : Index) return Boolean is
   begin
     if Memory (I).Is_Set then
       return Memory (I).Next = J;
     end if;
     return False;
   end Goes_To;

   procedure Link (I, J : Index) is
   begin
      Memory (I) := (Is_Set => True, Next => J);
   end Link;
end Lists;
```

Is this correct? 1/10

YES

```
package Lists with SPARK_Mode is
  function Goes_To (I, J : Index) return Boolean;
  procedure Link (I, J : Index) with Post => Goes_To (I, J);
private
  type Cell (Is_Set : Boolean := True) is record …
  type Cell_Array is array (Index) of Cell;
  Memory : Cell_Array;
end Lists;
package body Lists with SPARK_Mode is
   function Goes_To (I, J : Index) return Boolean is
   begin
     if Memory (I).Is_Set then
       return Memory (I).Next = J;
     end if;
     return False;
   end Goes_To;
   procedure Link (I, J : Index) is
   begin
      Memory (I) := (Is_Set => True, Next => J);
   end Link;
end Lists;
```

It is correct, but it cannot be verified with GNATprove. As Goes_To has no postcondition, nothing is known about its result

Is this correct? 2/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
package Lists with SPARK_Mode is
  function Goes_To (I, J : Index) return Boolean;

  procedure Link (I, J : Index) with Post => Goes_To (I, J);
private
  type Cell (Is_Set : Boolean := True) is record …
  type Cell_Array is array (Index) of Cell;

  Memory : Cell_Array;

  function Goes_To (I, J : Index) return Boolean is
      (Memory (I).Is_Set and then Memory (I).Next = J);
end Lists;

package body Lists with SPARK_Mode is
   procedure Link (I, J : Index) is
   begin
      Memory (I) := (Is_Set => True, Next => J);
   end Link;
end Lists;
```

Is this correct? 2/10

YES

```
package Lists with SPARK_Mode is
  function Goes_To (I, J : Index) return Boolean;

  procedure Link (I, J : Index) with Post => Goes_To (I, J);
private
  type Cell (Is_Set : Boolean := True) is record …
  type Cell_Array is array (Index) of Cell;

  Memory : Cell_Array;

  function Goes_To (I, J : Index) return Boolean is
      (Memory (I).Is_Set and then Memory (I).Next = J);
end Lists;

package body Lists with SPARK_Mode is
   procedure Link (I, J : Index) is
   begin
      Memory (I) := (Is_Set => True, Next => J);
   end Link;
end Lists;
```

Goes_To is an expression function. As a consequence, its body is available for proof.

Is this correct? 3/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
package Stacks with SPARK_Mode is
  type Stack is private;

  function  Peek (S : Stack) return Natural;
  procedure Push (S : in out Stack; E : Natural) with
    Post => Peek (S) = E;
private
  type Stack is record …
  function Peek (S : Stack) return Natural is
     (if S.Top in S.Content'Range then S.Content (S.Top) else 0);
end Stacks;

package body Stacks with SPARK_Mode is
  procedure Push (S : in out Stack; E : Natural) is
  begin
    if S.Top >= Max then
      return;
    end if;

    S.Top := S.Top + 1;
    S.Content (S.Top) := E;
  end Push;
end Stacks;
```

Is this correct? 3/10

NO

```
package Stacks with SPARK_Mode is
   type Stack is private;

   function  Peek (S : Stack) return Natural;
   procedure Push (S : in out Stack; E : Natural) with
     Post => Peek (S) = E;
private
   type Stack is record …
   function Peek (S : Stack) return Natural is
      (if S.Top in S.Content'Range then S.Content (S.Top) else 0);
end Stacks;

package body Stacks with SPARK_Mode is
   procedure Push (S : in out Stack; E : Natural) is
   begin
     if S.Top >= Max then
       return;
     end if;

     S.Top := S.Top + 1;
     S.Content (S.Top) := E;
   end Push;
end Stacks;
```

The postcondition of Push is only true if the stack is not full when Push is called.

Is this correct? 4/10

YES (click on the check icon)

NO (click on the error location(s))

```
package Stacks with SPARK_Mode is
  type Stack is private;

  function  Peek (S : Stack) return Natural;
  procedure Push (S : in out Stack; E : Natural) with
    Post => Peek (S) = E;
private
  type Stack is record …
  function Peek (S : Stack) return Natural is
     (if S.Top in S.Content'Range then S.Content (S.Top) else 0);
end Stacks;

package body Stacks with SPARK_Mode is
  procedure Push (S : in out Stack; E : Natural) is
  begin
    if S.Top >= Max then
      raise Is_Full_E;
    end if;

    S.Top := S.Top + 1;
    S.Content (S.Top) := E;
  end Push;
end Stacks;
```

Is this correct? 4/10

NO

```
package Stacks with SPARK_Mode is
  type Stack is private;

  function  Peek (S : Stack) return Natural;
  procedure Push (S : in out Stack; E : Natural) with
    Post => Peek (S) = E;
private
  type Stack is record …
  function Peek (S : Stack) return Natural is
     (if S.Top in S.Content'Range then S.Content (S.Top) else 0);
end Stacks;

package body Stacks with SPARK_Mode is
  procedure Push (S : in out Stack; E : Natural) is
  begin
    if S.Top >= Max then
      raise Is_Full_E;
    end if;

    S.Top := S.Top + 1;
    S.Content (S.Top) := E;
  end Push;
end Stacks;
```

GNATprove can now verify Push's postcondition as it only considers paths leading to normal termination. It will warn that Is_Full_E may be raised at runtime though, leading to an error.

Is this correct? 5/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
package Stacks with SPARK_Mode is
   type Stack is private;

   function  Peek (S : Stack) return Natural;
   function  Is_Full (S : Stack) return Natural;
   procedure Push (S : in out Stack; E : Natural) with
     Pre  => not Is_Full (S),
     Post => Peek (S) = E;
private
   type Stack is record …
   function Peek (S : Stack) return Natural is
      (if S.Top in S.Content'Range then S.Content (S.Top) else 0);
   function Is_Full (S : Stack) return Natural is (S.Top >= Max);
end Stacks;

package body Stacks with SPARK_Mode is
   procedure Push (S : in out Stack; E : Natural) is
   begin
     if S.Top >= Max then
       raise Is_Full_E;
     end if;
     S.Top := S.Top + 1;
     S.Content (S.Top) := E;
   end Push;
end Stacks;
```

Is this correct? 5/10

YES

```
package Stacks with SPARK_Mode is
   type Stack is private;

   function  Peek (S : Stack) return Natural;
   function  Is_Full (S : Stack) return Natural;
   procedure Push (S : in out Stack; E : Natural) with
     Pre  => not Is_Full (S),
     Post => Peek (S) = E;
private
   type Stack is record …
   function Peek (S : Stack) return Natural is
      (if S.Top in S.Content'Range then S.Content (S.Top) else 0);
   function Is_Full (S : Stack) return Natural is (S.Top >= Max);
end Stacks;

package body Stacks with SPARK_Mode is
   procedure Push (S : in out Stack; E : Natural) is
   begin
     if S.Top >= Max then
       raise Is_Full_E;
     end if;

     …
```

In the context of the precondition, GNATprove can now verify that Is_Full_E can never be raised at runtime.

Is this correct? 6/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
procedure Read_Record (From : Integer) is
  function Read_One (First : Integer; Offset : Integer)
    return Integer
  with
    Pre => Memory (First) + Offset in Memory'Range
  is
    Value : Integer := Memory (Memory (First) + Offset);
  begin
    if Is_Too_Coarse (Value) then
      Treat_Value (Value);
    end if;
    return Value;
  end Read_One;
begin
  Size := Read_One (From, 0);
  pragma Assume (Size in 1 .. 10
                 and then Memory (From) < Integer'Last - 2 * Size);
  Data1 := Read_One (From, 1);
  Addr  := Read_One (From, Size + 1);
  pragma Assume (Memory (Addr) > Memory (From) + Size);
  Data2 := Read_One (Addr, -Size);
end Read_Record;
```

Is this correct? 6/10

YES

```
procedure Read_Record (From : Integer) is
   function Read_One (First : Integer; Offset : Integer)
     return Integer
   with
     Pre => Memory (First) + Offset in Memory'Range
   is
     Value : Integer := Memory (Memory (First) + Offset);
   begin
     if Is_Too_Coarse (Value) then
       Treat_Value (Value);
     end if;
     return Value;
   end Read_One;
begin
   Size := Read_One (From, 0);
   pragma Assume (Size in 1 .. 10
                  and then Memory (From) < Integer'Last - 2 * Size);
   Data1 := Read_One (From, 1);
   Addr  := Read_One (From, Size + 1);
   pragma Assume (Memory (Addr) > Memory (From) + Size);
   Data2 := Read_One (Addr, -Size);
end Read_Record;
```

It is correct, but it cannot be verified with GNATprove. GNATprove analyses Read_One on its own and notices that an overflow may occur in its precondition in certain contexts.

Is this correct? 7/10

YES (click on the check icon)

NO (click on the error location(s))

```
procedure Read_Record (From : Integer) is
  function Read_One (First : Integer; Offset : Integer)
    return Integer
  with
    Pre => Memory (First) <= Memory'Last - Offset
  is
    Value : Integer := Memory (Memory (First) + Offset);
  begin
    if Is_Too_Coarse (Value) then
      Treat_Value (Value);
    end if;
    return Value;
  end Read_One;
begin
  Size := Read_One (From, 0);
  pragma Assume (Size in 1 .. 10
                 and then Memory (From) < Integer'Last - 2 * Size);
  Data1 := Read_One (From, 1);
  Addr  := Read_One (From, Size + 1);
  pragma Assume (Memory (Addr) > Memory (From) + Size);
  Data2 := Read_One (Addr, -Size);
end Read_Record;
```

Is this correct? 7/10

NO

```
procedure Read_Record (From : Integer) is
  function Read_One (First : Integer; Offset : Integer)
    return Integer
  with
    Pre => Memory (First) <= Memory’Last – Offset
  is
    Value : Integer := Memory (Memory (First) + Offset);
  begin
    if Is_Too_Coarse (Value) then
      Treat_Value (Value);
    end if;
    return Value;
  end Read_One;
begin
  Size := Read_One (From, 0);
  pragma Assume (Size in 1 .. 10
                and then Memory (From) < Integer'Last - 2 * Size);
  Data1 := Read_One (From, 1);
  Addr  := Read_One (From, Size + 1);
  pragma Assume (Memory (Addr) > Memory (From) + Size);
  Data2 := Read_One (Addr, -Size);
end Read_Record;
```

Unfortunately, our attempt to correct Read_One’s precondition failed. For example, an overflow will occur at runtime when Memory (First) is Integer'Last and Offset is negative.

Is this correct? 8/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
procedure Read_Record (From : Integer) is
  function Read_One (First : Integer; Offset : Integer)
    return Integer
  is
    Value : Integer := Memory (Memory (First) + Offset);
  begin
    if Is_Too_Coarse (Value) then
      Treat_Value (Value);
    end if;
    return Value;
  end Read_One;
begin
  Size := Read_One (From, 0);
  pragma Assume (Size in 1 .. 10
                 and then Memory (From) < Integer'Last - 2 * Size);
  Data1 := Read_One (From, 1);
  Addr  := Read_One (From, Size + 1);
  pragma Assume (Memory (Addr) > Memory (From) + Size);
  Data2 := Read_One (Addr, -Size);
end Read_Record;
```

Is this correct? 8/10

YES

```
procedure Read_Record (From : Integer) is
  function Read_One (First : Integer; Offset : Integer)
    return Integer
  is
    Value : Integer := Memory (Memory (First) + Offset);
  begin
    if Is_Too_Coarse (Value) then
      Treat_Value (Value);
    end if;
    return Value;
  end Read_One;
begin
  Size := Read_One (From, 0);
  pragma Assume (Size in 1 .. 10
                 and then Memory (From) < Integer'Last - 2 * Size);
  Data1 := Read_One (From, 1);
  Addr  := Read_One (From, Size + 1);
  pragma Assume (Memory (Addr) > Memory (From) + Size);
  Data2 := Read_One (Addr, -Size);
end Read_Record;
```

We could have fixed the contract on Read_One to handle correctly positive and negative values of Offset. However, we found it simpler to let the function be inlined for proof by removing its precondition.

Is this correct? 9/10

YES (click on the check icon)

NO (click on the error location(s))

```
procedure Compute (X : in out Integer) with
  Contract_Cases => ((X in -100 .. 100) => X = X'Old * 2,
                     (X in 0 .. 199)    => X = X'Old + 1,
                     (X in -199 .. 0)   => X = X'Old - 1,
                      X >= 200          => X = 200,
                      others            => X = -200)
is
begin
  if X in -100 .. 100 then
    X := X * 2;
  elsif X in 0 .. 199 then
    X := X + 1;
  elsif X in -199 .. 0 then
    X := X - 1;
  elsif X >= 200 then
    X := 200;
  else
    X := -200;
  end if;
end Compute;
```

Is this correct? 9/10

NO

```
procedure Compute (X : in out Integer) with
  Contract_Cases => ((X in -100 .. 100) => X = X'Old * 2,
                     (X in 0 .. 199)    => X = X'Old + 1,
                     (X in -199 .. 0)   => X = X'Old - 1,
                      X >= 200          => X = 200,
                      others            => X = -200)
is
begin
  if X in -100 .. 100 then
    X := X * 2;
  elsif X in 0 .. 199 then
    X := X + 1;
  elsif X in -199 .. 0 then
    X := X - 1;
  elsif X >= 200 then
    X := 200;
  else
    X := -200;
  end if;
end Compute;
```

We duplicated in Compute's contract the content of its body. This is not correct with respect to the semantics of Contract_Cases which expects disjoint cases, like a case statement.

Is this correct? 10/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
procedure Compute (X : in out Integer) with
  Contract_Cases => ((X in 1 .. 199)    => X >= X'Old,
                     (X in -199 .. -1) => X <= X'Old,
                      X >= 200          => X = 200,
                      X <= -200         => X = -200)
is
begin
  if X in -100 .. 100 then
    X := X * 2;
  elsif X in 0 .. 199 then
    X := X + 1;
  elsif X in -199 .. 0 then
    X := X - 1;
  elsif X >= 200 then
    X := 200;
  else
    X := -200;
  end if;
end Compute;
```

Is this correct? 10/10

NO

```
procedure Compute (X : in out Integer) with
  Contract_Cases => ((X in 1 .. 199)    => X >= X'Old,
                     (X in -199 .. -1) => X <= X'Old,
                      X >= 200          => X = 200,
                      X <= -200         => X = -200)
is
begin
  if X in -100 .. 100 then
    X := X * 2;
  elsif X in 0 .. 199 then
    X := X + 1;
  elsif X in -199 .. 0 then
    X := X - 1;
  elsif X >= 200 then
    X := 200;
  else
    X := -200;
  end if;
end Compute;
```

Here, GNATprove can successfully check that the different cases are disjoint. It can also successfully verify each case on its own. This is not enough though, as a Contract_Cases must also be total. Here, we forgot the value 0.

university.adacore.com