SPARK 2014

Quick Reference Examples

Aspects

```
aspect_specification ::=
  with aspect_mark [=> aspect_definition]
       {, aspect_mark [=> aspect_definition] }
```

SPARK Mode

```
package P
  with SPARK_Mode => On
is
   -- package spec is SPARK, so can be used
   -- by SPARK clients
end P;

package body P
  with SPARK_Mode => Off
is
   -- body is NOT SPARK, so assumed to
   -- be full Ada
end P;
```

Subprogram Contracts

PRECONDITIONS

```
function F (X : Integer) return Integer
  with Pre => X * X < 100;

procedure P (X : Integer; Y : Integer)
  with Pre => X + Y = 0 and then F (Y) /= 0;

procedure Some_Call
  with Pre => Initialized; --  before it is declared
Initialized : Boolean := False;
```

POSTCONDITIONS

```
procedure Increment (X : in out Integer)
  with Pre  => X < Integer'Last,
       Post => X = X'Old + 1;
```

CONTRACT CASES

```
procedure Bounded_Add
  (X, Y : in Integer; Z : out Integer)
  with Contract_Cases =>
    ((X + Y in Integer'Range) => Z = X + Y,
     Integer'First > X + Y    => Z = Integer'First,
     X + Y > Integer'Last     => Z = Integer'Last);
```

GLOBAL CONTRACTS

```
procedure P
  with Global => (Input    => (A, B, C),
                  Output   => (L, M, N),
                  In_Out   => (X, Y, Z),
                  Proof_In => (I, J, K));
```

DEPENDS CONTRACTS

Contracts for information flow analysis.

```
procedure Sum
  (A, B : in Integer; Result : out Integer)
  with Depends => (Result => (A, B));
```

\+ indicates self -dependency

```
procedure Update_Array (A : in out Array_Type;
                        I : in     Index_Type;
                        X : in     Elem_Type)
  with Depends => (A => +(I, X));

procedure Clear_Stack (S : out Stack)
  with Depends => (S => null);

procedure P (X, Y, Z : in T)
  with Depends => (null => (X, Y, Z));
```

ASSUME

No verification condition generated - soundness alert!
Use with great care.

```
...
pragma Assume (Ticks < Time_Type'Last);
...
```

LOOP INVARIANT

```
pragma Loop_Invariant
  (J in Low .. High and
   (for all K in Low .. J => not Is_Prime (K)));
```

LOOP VARIANT

```
pragma Loop_Variant (Increases => I,
                     Decreases => F (X));
```

LOOP ENTRY

```
type Array_T is
  array (1 .. 10) of Integer range 0 .. 7;
...
for I in A'Range loop
   Result := Result + A (I);
   pragma Loop_Invariant
     (Result <= Result'Loop_Entry + 7 * I);
end loop;
```

Expressions

Expressions that are particularly useful when writing contracts

IF EXPRESSIONS

```
A := (if X then 2 else 3);
```

CASE EXPRESSIONS

```
B := (case Y is
        when E1 => V1,
        when E2 => V2,
        when others => V3);
```

BOOLEAN SHORT-CIRCUIT OPERATORS

```
function F (X, Y : Integer) return Integer
  with Pre => (Y /= 0 and then X/Y > Limit);


function G (X, Y : Integer) return Integer
  with Pre => (Y /= 0 or else (X/Y) /= 10);
```

QUANTIFIED EXPRESSIONS

```
procedure Set_Array (A: out Array_Type)
  with Post => (for all M in A'Range => A(M) = M);


function Contains (A   : Array_Type;
                   Val : Element_Type) return Boolean
  with Post => (for some J in A'Range => A(J) = Val);
```

EXPRESSION FUNCTIONS

```
function Value_Found_In_Range
  (A       : Arr;
   Val     : Element;
   Low, Up : Index) return Boolean
  is (for some J in Low .. Up => A(J) = Val);


function Add_One (X : in Integer) return Integer
  is (X + 1)
  with Pre => (X < Integer'Last);
```

'RESULT

```
package Find is
   type A is array (1..10) of Integer;
   function Find (T : A; R : Integer) return Integer
      with Post => Find'Result >= 0 and then
         (if Find'Result /= 0 then T(Find'Result) = R);
end Find;
```

'UPDATE EXPRESSIONS

```
procedure P (R : in out Rec)
  with
Post => R = R'Old'Update (X => 1, Z => 5);


A1 := Some_Array'Update (1 .. 10 => True,
                         5       => False);


A2 := Some_Array'Update (Param_1'Range => True,
                         Param_2'Range => False);
```

'OLD EXPRESSIONS

```
procedure Increment (X : in out Integer)
  with Post => X = X'Old + 1;

Some_Global : Integer;

procedure Call_Not_Modify_Global
  with Post => Some_Global =
               Some_Global'Old;

type T is record
   A : Integer;
   B : Integer
end record;

function F (V : T) return Integer;

procedure P (V : in out T)
  with Post => V'Old.A   /= V.A   and then
               V.B'Old   /= V.B   and then
               F (V'Old) /= F (V) and then
               F (V)'Old /= F (V);

pragma Unevaluated_Use_Of_Old (Allow);
--  to allow Expr'Old when Expr not variable,
--  in context not always evaluated
```

Package Contracts

ABSTRACT STATE

```
package P
   with Abstract_State =>
        (Essential_State, Result_Cache)
        -- Parentheses not requied
        -- if only one state abstraction
is
   ...
end P;
```

REFINED STATE

```
package body P
   with Refined_State =>
        (Essential_State => (E1, E2),
         Result_Cache  => Cache)
is
   ...
end P;
```

INITIALIZATION

```
package A_Stack
   with Abstract_State     => Stack,
        Initializes        => Stack,
        Initial_Condition => Stack_Empty
        -- state abstractions are not listed
        -- in an Initializes contract if they are
        -- not initialized by package elaboration
is
   function Stack_Empty return Boolean
     with Global => Stack;
    ...
end A_Stack;

package Three_States
   with Abstract_State =>
             (State_1,
              State_2,
              Unintitialized_State),
        Initializes    =>
            (State_1, State_2)
is
   ...
end Three_States;
```

EXTERNAL STATE

```
with System.Storage_Elements;

package Output_Port
is
   Sensor : Integer
     with Volatile,
          Async_Readers,
          Address =>
          System.Storage_Elements.To_Address
              (16#ACECAFE#);
end Output_Port;

package Abstract_Input_Device
   with Abstract_State =>
              (Input_Dev with External =>
                 (Async_Writers, Effective_Reads)),
        Initializes => Input_Dev
is
   ...
end Abstract_Input_Device;
```

PART_OF

```
package P
   with Abstract_State => State_P
is
   ...
private
   Hidden_Var : Integer with
      Part_Of => State_P;
   ...
end P;

...
package Q
   with Abstract_State => (S1, S2),
        Initializes => S1
is
   ...
end Q;

private package Q.Child
   with Abstract_State =>
             (Child_State with Part_Of => Q.S1),
        Initializes => Child_State
is
   ...
end Q.Child;
```

Warnings and Check Message Control

```
package body Warnings_Example is
   pragma Warnings
      (Off, "formal parameter ""X"" is not referenced");
   procedure Mumble (X : Integer) is
   pragma Warnings
      (On, "formal parameter ""X"" is not referenced");
    -- X is ignored here, because ... etc.
   begin
      null;
   end Mumble;
end Warnings_Example;
```

Remember that every failed check message corresponds to a soundness issue and should be reviewed / justified individually.

```
return (X + Y) / (X - Y);
pragma Annotate
  (GNATprove, False_Positive,
   "divide by zero", "reviewed by John Smith");


procedure Do_Something (X, Y : in out Integer) with
  Depends => ((X, Y) => (X, Y));
pragma Annotate
  (GNATprove, Intentional,
   "incorrect dependency ""Y => X""",
   "Dependency is kept for compatibility reasons");
```

Flags for the SPARK Tools

Options for the compiler and GNATprove.

OVERFLOW CHECKING MODES

GNAT Pro compiler switch controls semantics of overflow checks in assertions (contracts) and code. Three modes:

1 = strict Ada semantics for overflow checking

2 = minimized overflow checking

3 = eliminated - no possibility of overflow (mathematical semantics)

Example: **-gnato13**

- First digit specifies overflow mode for code.
- Second digit specifies overflow mode for contracts.