Programming by Contract

José F. Ruiz
Senior Software Engineer

Ada Spain 2013


Ada 2012

Ada Evolution

Ada 83

Ada 95

- Object Orientation
- Better Access Types
- Protected Types
- Child Packages

Ada 2005

- Interfaces
- Containers
- Better Limited Types
- Ravenscar

Ada 2012

- Pre / Post / Invariants
- Iterators
- New expressions
- Process Affinities

In out parameters for functions

- Ada 83 to 2005 forbids the use of in out for function
- Since Ada 95, it's possible to workaround that with the access mode (but requires the explicit use of an access)
- Ada 2012 allows 'in out' parameters for functions

```
function Increment (V : in out Integer) return Integer is
begin
   V := V + 1;
   return V;
end F;
```

Aliasing detection

- **Ada 2012 detects "obvious" aliasing problems**

```
 function Change (X, Y : in out Integer) return Integer is
   begin
      X := X * 2;
      Y := Y * 4;

      return X + Y;
   end;

   One, Two : Integer := 1;

begin

   Two := Change (One, One);
   -- warning: writable actual for "X" overlaps with actual for "Y"

   Two := Change (One, Two) + Change (One, Two);
   --  warning: result may differ if evaluated after other actual in expression
```

Pre, Post conditions and Invariants

- **The Ada 2012 standard normalizes pre conditions, post conditions**

```
procedure P (V : in out Integer)
   with Pre  => V >= 10,
        Post => V'Old /= V;
```

- **New type invariants will ensure properties of an object**

```
type T is private
   with Invariant => Check (T);
```

- **Subtype predicates**

```
type Even is range 1 .. 10
   with Predicate => Even mod 2 = 0;
```

Conditional expressions

- It will be possible to write expressions with a result depending on a condition

```
procedure P (V : Integer) is
   X : Integer := (if V = 10 then 15 else 0);
   Y : Integer := (case V is when 1 .. 10 => 0, when others => 10);
begin
   null;
end;
```

Iterators

- Given a container, it will be possible to write a simple loop iterating over the elements
- Custom iterators will be possible

Ada 2005

```
   X : Container.Iterator := First (C);
   Y : Element_Type;
declare
   while X /= Container.No_Element loop
      -- work on X
      Y := Container.Element (X);
      -- work on Y
      X := Next (X);
   end loop;
```

Ada 2012

```
for X in C loop
   -- work on X
   Y := Container.Element (X);
   -- work on Y
end loop;
```

```
for Y of C loop
   -- work on Y
end loop;
```

Quantifier expressions

- **Checks that a property is true on all components of a collection (container, array…)**

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

V  : A := (10, 20, 30);
B1 : Boolean := (for all J in V'Range => V (J) >= 10);  -- True
B2 : Boolean := (for some J in V'Range => V (J) >= 20); -- True
```

Generalized memberships tests

- **Memberships operations are now available for all kind of Boolean expressions**

Ada 2005

```
if C = 'a'
   or else C = 'e'
   or else C = 'i'
   or else C = 'o'
   or else C = 'u'
   or else C = 'y'
then
```

```
case C is
   when 'a' | 'e' | 'i'
      | 'o' | 'u' | 'y' =>
```

Ada 2012

```
if C in 'a' | 'e' | 'i'
      | 'o' | 'u' | 'y' then
```

Expression-functions

- **Function implementation can be directly given at specification time if it represents only an "expression"**

```
function Even (V : Integer) return Boolean
   is (V mod 2 = 0);
```

Containers

- **Ada 2005 containers are unsuitable for HIE application**
  - Rely a lot of the runtime
  - Not bounded

- **Ada 2012 introduces a new form of container, "bounded" used for**
  - HIE product
  - Statically memory managed
  - Static analysis and proof

Processor affinities

- Task can be assigned to specific processors
- Enhances control over program behavior
- Enables Ravenscar on multi-core

```
task body T1 is
   pragma CPU (1);
begin
   […]
end T1;

task body T2 is
   pragma CPU (2);
begin
   […]
end T2;
```

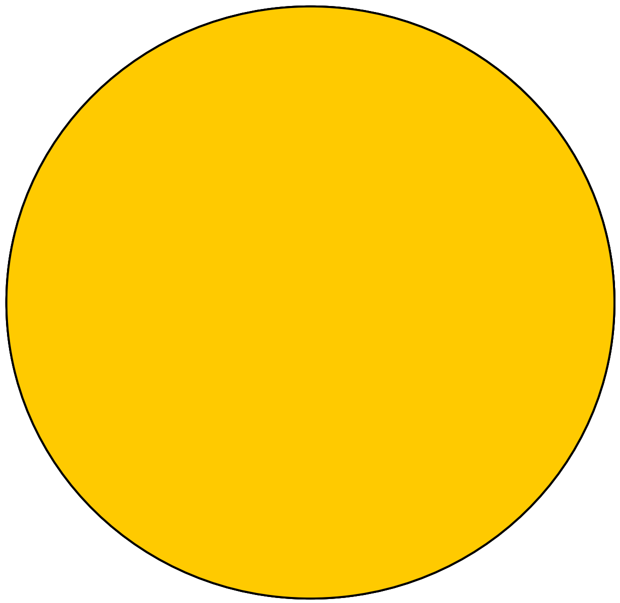
Ada 2012 safety improvements

- **Improve readability**
  - Specification contains formally expressed properties on the code
- **Improve testability**
  - Constraints on subprograms & code can lead to dynamic checks enabled during testing
- **Allow more static analysis**
  - The compiler checks the consistency of the properties
  - Static analysis tools (CodePeer) uses these properties as part of its analysis
- **Allow more formal proof**
  - Formal proof technologies can prove formally certain properties of the code (High-Lite project)

T

SPARK 2014

- Testing is expensive and inaccurate
- Proving is more accurate, but proving 100% is even more expensive…

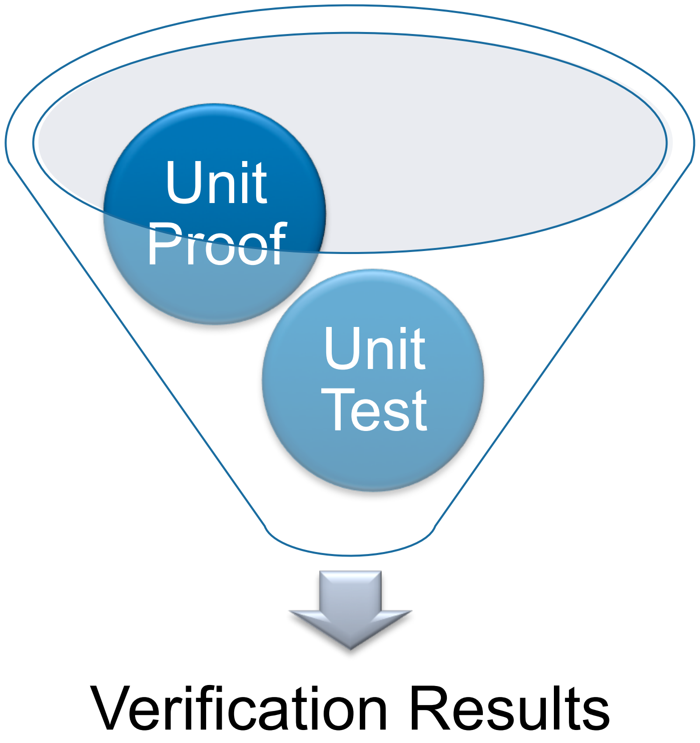


- … especially the last 20%
- How about proving what's easy to prove and test the rest?

Principles

- **SPARK 2014 is a subset of Ada**
  - Provable subset (Ada without tasking, exceptions and access / aliasing)
  - Can act as a coding standard for e.g. DO-178B
  - Provable
- **Proofs are made against formal contract (pre / post conditions)**
- **Sometimes it's not practical to**
  - Write in SPARK 2014
  - Write the contracts
  - Prove the code
- **Test can replace proof**
- **Objective : be at least as good as test, at most as expensive as tests**

Combining Unit Proof and Unit Test



Unit Proof is done by SPARK toolset, relying on provers (e.g. Alt-Ergo)

Unit test can be done either using GNATtest or standard testing technology

Proof with SPARK 2014

- Absence of Run-Time errors (exceptions)
- Each point of call verifies the preconditions of this subprogram
- Each subprograms verifies the postcondition assuming that the precondition is true
- Predicates and type invariants are verified on type usage
- Addition of constructions dedicated to proof (loop invariants, object update, …)

Example
Programming by Contract

Example: a ring buffer

DATA

```
Buf_Size : constant := 100;

type Buf_Array is array (0 .. Buf_Size - 1) of Float;
-- The array which stores the buffer

type Ring_Buffer is record
   Data   : Buf_Array := (others => 0.0);
   First  : Integer   := 0;
   Length : Integer   := 0;
end record;
-- The record representing the buffer.
-- First is the first cell containing valid data.
-- Length is the number of stored items.
-- Wrapping around the array borders is possible.

-- The field Length is between 0 and Buf_Size.
-- The field First is always a valid array index, hence
-- between 0 and Buf_Size - 1.
```

Example: a ring buffer (II)

API

```
function Is_Empty (R : Ring_Buffer) return Boolean;
--  Check whether the buffer is empty

function Is_Full (R : Ring_Buffer) return Boolean;
--  Check whether the buffer is full

function Head (R : Ring_Buffer) return Float;
--  Return the first element of the buffer

procedure Push (R : in out Ring_Buffer; Element : Float);
--  Insert element in the buffer. The buffer should not be full
--  and its length is increased by one.

procedure Pop (R : in out Ring_Buffer; Element : out Float);
--  Extract the first element of the buffer. The buffer should
--  not be empty and its length is decreased by one.
```

Enhance ring buffer: better typing

```
Buf_Size : constant := 100;

type Length_Type is new Integer range 0 .. Buf_Size;
-- The integer type of buffer length

type Index_Type is mod Length_Type'Last;
-- The integer type for valid array indices

type Buf_Array is array (Index_Type) of Float;

type Ring_Buffer is record
   Data   : Buf_Array   := (others => 0.0);
   First  : Index_Type  := 0;
   Length : Length_Type := 0;
end record;
```

Enhance ring buffer: use Ada 2012 expression functions

```
function Is_Empty (R : Ring_Buffer) return Boolean is
   (R.Length = 0);
--  Check whether the buffer is empty

function Is_Full (R : Ring_Buffer) return Boolean is
   (R.Length = Buf_Size);
--  Check whether the buffer is full

function Head (R : Ring_Buffer) return Float is
   (R.Data (R.First));
--  Return the first element of the buffer
```

Enhance ring buffer: contracts

```
procedure Push (R : in out Ring_Buffer; Element : Float) with
   Pre  => not Is_Full (R),
   Post => R.Length = R.Length'Old + 1;
--  Insert element in the buffer. The buffer should not be full
--  and its length is increased by one.

procedure Pop (R : in out Ring_Buffer; Element : out Float) with
   Pre  => not Is_Empty (R),
   Post => R.Length = R.Length'Old - 1 and then
           R.First = R.First'Old + 1 and then
           Head (R'Old) = Element;
--  Extract the first element of the buffer. The buffer should
--  not be empty and its length is decreased by one.
```

What can we do with contracts?

Possibilities

- **Static verification**
  - The compiler has limited checks
    - Must run quickly -> imprecise analysis
    - Can detect “obvious” errors
  - Verifier performs longer and better analysis
    - Longer execution -> precise analysis
    - Scalable analysis -> modular, based on contracts
    - Can detect subtle errors

- **Run-time checks**
  - Contracts behave like Assertions

- **Formal proofs**
  - SPARK 2014

Static verification: compiler

```
function Increase (X : Integer) return Integer with
     Post => X < Integer'Last;
```

```
$ gcc -c -gnat12 -gnata pck.adb
warning: postcondition refers only to pre-state
warning: function postcondition does not mention result
```

Verifier

- **The Verifier checks**
  - All possible run-time errors
    - Division by zero, range checks, …
  - All user properties
    - Assertions
    - Contracts
    - Invariants

- **The Verifier works by**
  - Generating specific logical formulas
    - Called Verification Conditions (VCs)
  - Using a prover to verify them

- **Strong mathematical origins**

Static verification: verifier (postcondition)

```
procedure Push (R : in out Ring_Buffer; Element : Float) with
   Pre  => not Is_Full (R),
   Post => R.Length = R.Length'Old + 1;

procedure Pop (R : in out Ring_Buffer; Element : out Float) with
   Pre  => not Is_Empty (R),
   Post => R.Length = R.Length'Old - 1 and then
           R.First = R.First'Old + 1 and then
           Head (R'Old) = Element;
```

```
$ gnatprove -v --report=detailed -P default.gpr
analyzing Example_2012.Push, 3 checks
example_2012.adb:9:38: info: range check proved
example_2012.adb:10:28: info: range check proved
example_2012.ads:37:21: info: postcondition proved
analyzing precondition for Example_2012.Push, 0 checks
analyzing Example_2012.Pop, 2 checks
example_2012.adb:20:28: info: range check proved
example_2012.ads:43:21: info: postcondition proved
analyzing precondition for Example_2012.Pop, 0 checks
```

Static verification: verifier (range checks)

```
procedure Push (R : in out Ring_Buffer; Element : Float) is
begin
   R.Data (R.First + Index_Type (R.Length)) := Element;
   R.Length := R.Length + 1;
end Push;

procedure Pop (R : in out Ring_Buffer; Element : out Float) is
begin
   Element := R.Data (R.First);
   R.Length := R.Length - 1;
   R.First := R.First + 1;
end Pop;
```

```
example_2012.adb:9:38: info: range check proved
example_2012.adb:10:28: info: range check proved
example_2012.ads:37:21: info: postcondition proved
analyzing precondition for Example_2012.Push, 0 checks
analyzing Example_2012.Pop, 2 checks
example_2012.adb:20:28: info: range check proved
example_2012.ads:43:21: info: postcondition proved
analyzing precondition for Example_2012.Pop, 0 checks
```

What if a VC is not proved?

- **Causes**
  - Incorrect code
  - Incorrect assertion
  - Missing assertions about program behavior
  - Prover timeouts
  - Prover not smart enough

- **How to investigate**
  - Relatively easy
    - Pre/post conditions, assertions, and invariants are executable
      - You can run and debug them
    - Increase prover timeout
    - Use alternative SMT prover
  - Time consuming
    - Manual review
  - Time consuming and difficult
    - Hand-written proofs

  - … or testing

SPARK 2014

SPARK 2014 language (I)

- **Completely based on Ada 2012**
  - New specification aspects: contracts, invariants
  - New expresions: if expression, case expression, quantified expression (for all, for some)
  - New attributes: ‘Result, ‘Old

```
--  Old SPARK
procedure Inc (X : in out Integer);
--# pre X < Integer'Last;
--# post X = X~ + 1;

-- New SPARK 2014
procedure Inc (X : in out Integer) with
   Pre  => X < Integer'Last,
   Post => X = X'Old + 1;
```

SPARK 2014 language (II)

- **Main restrictions with respect to Ada**
  - Functions cannot have side-effects
  - No pointers (no access types)
  - No aliasing (between references)
  - No exceptions
  - No tasking

- **Additional constructs specific to SPARK 2014**
  - New aspects: *Contract_Cases*, *Global*, *Depends*
  - New pragmas: *Loop_Invariant*, *Loop_Variant*
  - New attributes: *Loop_Entry*, *Update*

SPARK 2014 example

```
type Index is range 1 .. 10;
type Elements is range 0 .. 100;
type Elt_Array is array (Index) of Elements;

function Max (E1, E2 : Elements) return Elements is
   (if E1 < E2 then E2 else E1);

procedure Max_Array (A : Elt_Array; EMax : out Elements) with
   Global  => null,
   Depends => (Emax => A),
   Post    => (for all Elt of A => EMax >= Elt);

procedure Max_Array (A : Elt_Array; EMax : out Elements) is
begin
   EMax := Elements'First;
   for J in Index loop
      if A (J) > EMax then
        EMax := A (J);
      end if;
      pragma Loop_Invariant
         (EMax >= Emax’Loop_Entry and
          (for all K in Index’First .. J => (EMax >= A (K))));
   end loop;
end Max_Array;
```