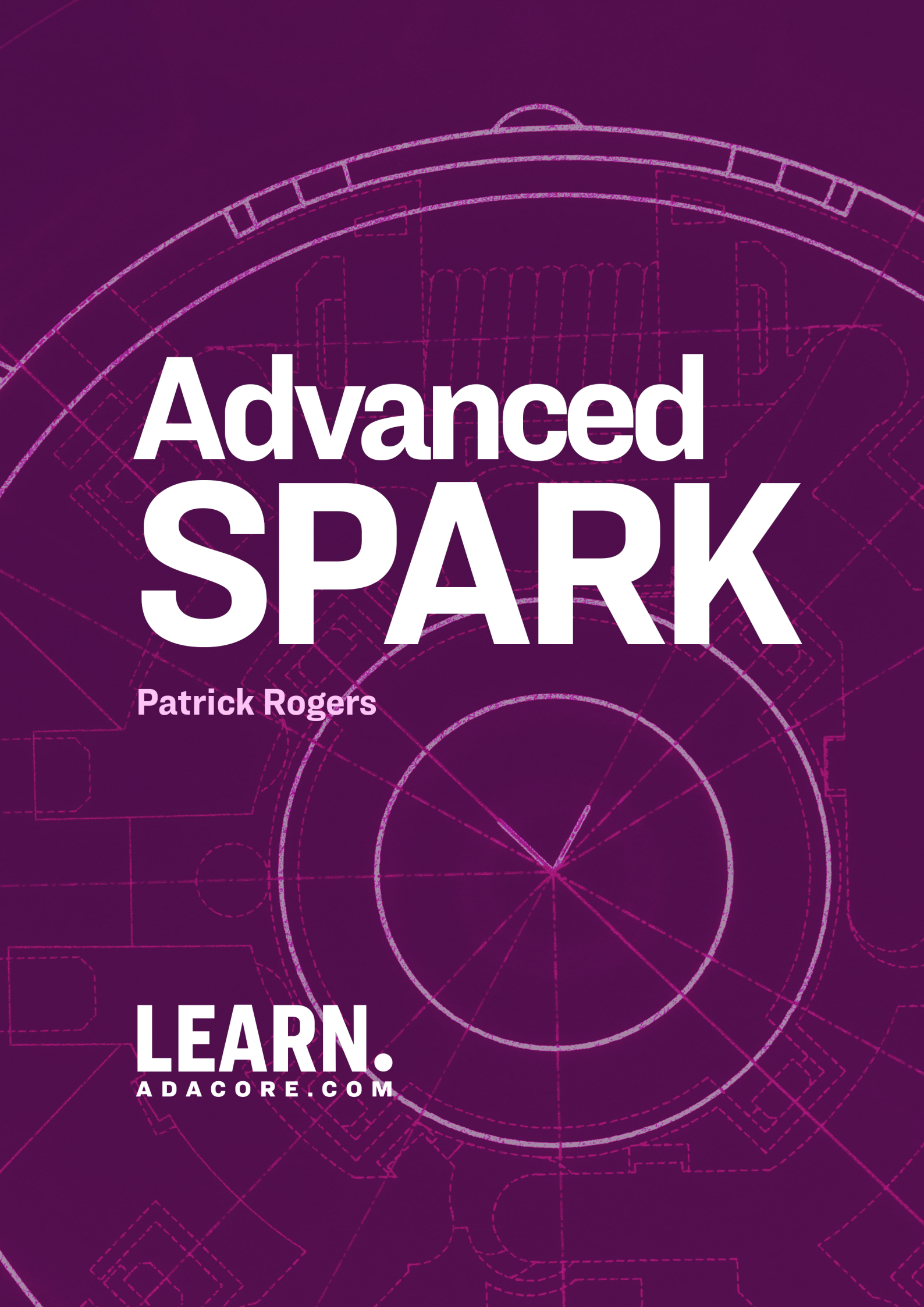


Advanced SPARK

Patrick Rogers

LEARN.
ADACORE.COM

LEARN.
ADACORE.COM

Advanced SPARK

Release 2024-09

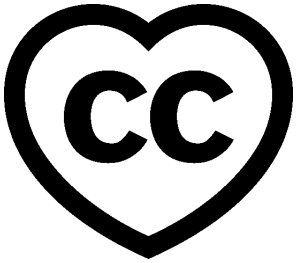
Patrick Rogers

Sep 08, 2024

CONTENTS:

Warning

This version of the website contains UNPUBLISHED contents. Please do not share it externally!



This course will teach you advanced topics of SPARK.

Note

The code examples in this course use an 80-column limit, which is a typical limit for Ada code. Note that, on devices with a small screen size, some code examples might be difficult to read.

Note

Each code example from this book has an associated "code block metadata", which contains the name of the "project" and an MD5 hash value. This information is used to identify a single code example.

You can find all code examples in a zip file, which you can download from the learn website[2]. The directory structure in the zip file is based on the code block metadata. For example, if you're searching for a code example with this metadata:

- Project: Courses.Intro_To_Ada.Imperative_Language.Greet
- MD5: cba89a34b87c9dfa71533d982d05e6ab

you will find it in this directory:

```
projects/Courses/Intro_To_Ada/Imperative_Language/Greet/
cba89a34b87c9dfa71533d982d05e6ab/
```

In order to use this code example, just follow these steps:

1. Unpack the zip file;
2. Go to target directory;
3. Start GNAT Studio on this directory;
4. Build (or compile) the project;

[1] http://creativecommons.org/licenses/by-sa/4.0

5. Run the application (if a main procedure is available in the project).

[2] https://learn.adacore.com/zip/learning-ada_code.zip

CHAPTER

ONE

SUBPROGRAM CONTRACTS

1.1 Subprogram Contracts in Ada 2012 and SPARK 2014

- Originate in Floyd-Hoare logic (1967-1969)
  - a Hoare triple {P}C{Q}
  - P is the precondition before executing command C
  - Q is the postcondition after executing command C
- Executable version by Meyer in Eiffel (1988)
  - Called Design by Contract ™
  - Precondition is checked dynamically before a routine starts
  - Postcondition is checked dynamically when a routine returns
- SPARK 2014 combines both views
  - SPARK 2005 version was only logic, Ada version is only executable

1.2 Dynamic Execution of Subprogram Contracts

- Contract on subprogram declaration
  - Different from subprogram body in general (but not always)
- Ada Reference Manual allows implementations choice
  - Contract can be checked in the caller or in the callee
  - GNAT's choice is to execute in the callee
- GNAT introduces wrappers in some cases for contracts
  - For an imported subprogram (e.g. from C) with a contract
  - For cases where contracts on static call/dispatching are different
- Contracts are not enabled by default
  - Switch `-gnata` enables dynamic checking of contracts in GNAT

1.3 Dynamic Behavior when Subprogram Contracts Fail

- Violation of contract raises an exception
  - Standard exception `Assertion_Error` is raised (same as for pragma `Assert` and all other assertions)
  - Exception cannot be caught by subprogram's own exception handler implementation choice caller/callee has no effect
  - Idiom allows to select another exception

Listing 1: show_dynamic_behavior.ads

```
with Ada.Numerics; use Ada.Numerics;

package Show_Dynamic_Behavior is

   function Sqrt (X : Float) return Float with
      Pre => X >= 0.0 or else raise Argument_Error;

end Show_Dynamic_Behavior;
```

- Control over sequencing of checks
  - Typical pre/post is a conjunction of Boolean conditions
  - Use and when no possible RTE, and then otherwise (recommended for SPARK)

1.4 Precondition

- Better alternative to defensive programming, compare

Listing 2: show_precondition.ads

```
with Ada.Numerics; use Ada.Numerics;

package Show_Precondition is

   function Sqrt (X : Float) return Float with
      Pre => X >= 0.0 or else raise Argument_Error;

end Show_Precondition;
```

and

Listing 3: show_precondition.ads

```
with Ada.Numerics; use Ada.Numerics;

package Show_Precondition is

   --  X should be non-negative or Argument_Error is raised
   function Sqrt (X : Float) return Float;

end Show_Precondition;
```

Listing 4: show_precondition.adb

```
package body Show_Precondition is

   function Sqrt (X : Float) return Float is
      Res : Float := 0.0;
   begin
      if X >= 0.0 then
         raise Argument_Error;
      end if;

      --  [...]

      return Res;
   end Sqrt;

end Show_Precondition;
```

- Preconditions can be activated alone

```
pragma Assertion_Policy (Pre => Check);
```

1.5 Postcondition

- Single place to check all return paths from the subprogram
  - Avoids duplication of checks before each return statement
  - Much more robust during maintenance
  - Only applies to normal returns (not in exception, not on abort)
- Can relate input and output values
  - Special attribute `X'Old` for referring to input value of variable X
  - Special attribute `Func'Result` for referring to result of function Func
  - Special attribute `Rec'Update` or `Arr'Update` for referring to modified value of record Rec or array Arr
    - replaced by delta aggregate syntax in Ada 202X: `( Rec with delta Comp => Value)`

1.6 Contract Cases

- Convenient syntax to express a contract by cases
  - Cases must be disjoint and complete (forming a partition)
  - Introduced in SPARK, planned for inclusion in Ada 202X
  - Case is (guard => consequence) with `'Old` / `'Result` in consequence
  - Can be used in combination with precondition/postcondition

Listing 5: show_contract_cases.ads

```
package Show_Contract_Cases is

   function Sqrt (X : Float) return Float with
     Contract_Cases =>
       (X > 1.0             => Sqrt'Result <= X,
        X = 1.0             => Sqrt'Result = 1.0,
        X < 1.0 and X > 0.0 => Sqrt'Result >= X,
        X = 0.0             => Sqrt'Result = 0.0);

end Show_Contract_Cases;
```

- Both a precondition and a postcondition
  - On subprogram entry, exactly one guard must hold
  - On subprogram exit, the corresponding consequence must hold

1.7 Attribute 'Old

- X'Old expresses the input value of X in postconditions
  - Same as X when variable not modified in the subprogram
  - Compiler inserts a copy of X on subprogram entry if X is large, copy can be expensive in memory footprint!
  - X can be a variable, a function call, a qualification (but not limited!)

Listing 6: show_attribute_old.ads

```
package Show_Attribute_Old is

   type Value is new Integer;

   type My_Range is range 1 .. 10;

   type My_Array is array (My_Range) of Value;

   procedure Extract (A : in out My_Array;
                      J :        My_Range;
                      V :    out Value)
     with
       Post => (if J in A'Range then V = A (J)'Old and A (J) = 0);

end Show_Attribute_Old;
```

- Expr'Old is rejected in potentially unevaluated context
  - **pragma** *Unevaluated_Use_Of_Old* (Allow) allows it
  - In Ada, user is responsible – in SPARK, user can rely on proof

1.8 Implication and Equivalence

- If-expression can be used to express an implication
  - `(if A then B)` expresses the logical implication
    - `A → B`
  - `(if A then B else C)` expresses the formula
    - `(A → B) (¬A → C)`
  - `(if A then B else C)` can also be used with B, C not of Boolean type
  - `(A <= B)` should not be used for expressing implication (same dynamic semantics, but less readable, and harmful in SPARK)
- Equality can be used to express an equivalence
  - `(A = B)` expresses the logical equivalence
    - `(A ↔ B)`
  - A double implication should not be used for expressing equivalence (same semantics, but less readable and maintainable)

1.9 Reasoning by Cases

- Case-expression can be used to reason by cases
  - Case test only on values of expressions of discrete type
  - Can sometimes be an alternative to contract cases

Listing 7: show_case_expression.ads

```
with Ada.Text_IO;

package Show_Case_Expression is

   type File_Mode is (Open, Active, Closed);

   type File is record
      F_Type : Ada.Text_IO.File_Type;
      Mode   : File_Mode;
   end record;

   procedure Open (F : in out File; Success : out Boolean) with
     Post =>
       (case F.Mode'Old is
          when Open   => Success,
          when Active => not Success,
          when Closed => Success = (F.Mode = Open));

end Show_Case_Expression;
```

- Can sometimes be used at different levels in the expression

```
procedure Open (F : in out File; Success : out Boolean) with
  Post =>
    Success = (case F.Mode'Old is
                 when Open   => True,
```

(continues on next page)

(continued from previous page)

```
                when Active => False,
                when Closed => F.Mode = Open);
```

1.10 Universal and Existential Quantification

- Quantified expressions can be used to express a property over a collection of values
  - `(for all X in A .. B => C)` expresses the universally quantified property
    - $(\forall X . X \geq A \square X \leq B \rightarrow C)$
  - `(for some X in A .. B => C)` expresses the universally quantified property
    - $(\exists X . X \geq A \square X \leq B \square C)$
- Quantified expressions translated as loops at run time
  - Control exits the loop as soon as the condition becomes false (resp. true) for a universally (resp. existentially) quantified expression
- Quantification forms over array and collection content
  - Syntax uses `(for all/some V of ... => C)`

1.11 Expression Functions

- Without abstraction, contracts can become unreadable
  - Also, use of quantifications can make them unprovable
- Expression functions provide the means to abstract contracts
  - Expression function is a function consisting in an expression
  - Definition can complete a previous declaration
  - Definition is allowed in a package spec! (crucial for proof with SPARK)

```
function Valid_Configuration return Boolean is
   (case Cur_State is
      when Piece_Falling | Piece_Blocked =>
        No_Overlap (Cur_Board, Cur_Piece),
      when Board_Before_Clean => True,
      when Board_After_Clean =>
        No_Complete_Lines (Cur_Board));
```

1.12 Code Examples / Pitfalls

1.12.1 Example #1

Listing 8: example_01.adb

```
with Ada.Assertions; use Ada.Assertions;

procedure Example_01 is

   --  Fail systematically fails a precondition and catches the
   --  resulting exception.

   procedure Fail (Condition : Boolean) with
     Pre => Condition
   is
      Bad_Condition : Boolean := False;
   begin
      Fail (Bad_Condition);
   exception
      when Assertion_Error => return;
   end Fail;
begin
   null;
end Example_01;
```

This code is not correct. The exception from the recursive call is always caught in the handler, but not the exception raised if caller of `Fail` passes **`False`** as value for `Condition`.

1.12.2 Example #2

Listing 9: example_02.ads

```
with Interfaces.C; use Interfaces.C;

package Example_02 is

   procedure Memset
     (B  : in out char_array;
      Ch :        char;
      N  :        size_t)
     with
       Import,
       Pre  => N <= B'Length,
       Post => (for all Idx in B'Range =>
                  (if Idx < B'First + N then
                         B (Idx) = Ch
                       else
                         B (Idx) = B'Old (Idx)));

end Example_02;
```

This code is correct. GNAT will create a wrapper for checking the precondition and postcondition of `Memset`, calling the imported `memset` from `libc`.

1.12.3 Example #3

Listing 10: example_03.adb

```
procedure Example_03 is

   pragma Assertion_Policy (Pre => Ignore);
   function Sqrt (X : Float) return Float with
     Pre => X >= 0.0;

   pragma Assertion_Policy (Pre => Check);
   function Sqrt (X : Float) return Float is
      Ret : Float := 0.0;
   begin
      --  missing implementation...
      return Ret;
   end Sqrt;

begin
   null;
end Example_03;
```

This code is not correct. Although GNAT inserts precondition checks in the subprogram body instead of its caller, it is the value of Pre assertion policy at the declaration of the subprogram that decides if preconditions are activated.

1.12.4 Example #4

Listing 11: example_04.adb

```
procedure Example_04 is

   function Sqrt (X : Float) return Float with
     Pre => X >= 0.0;

   function Sqrt (X : Float) return Float with
     Pre => X >= 0.0
   is
      Ret : Float := 0.0;
   begin
      --  missing implementation...
      return Ret;
   end Sqrt;

begin
   null;
end Example_04;
```

This code is not correct. Contract is allowed only on the spec of a subprogram. Hence it is not allowed on the body when a separate spec is available.

1.12.5 Example #5

Listing 12: example_05.adb

```
procedure Example_05 is

   procedure Add (X, Y : Natural; Z : out Integer) with
     Contract_Cases =>
       (X <= Integer'Last - Y => Z = X + Y,
        others                => Z = 0)
   is
   begin
      Z := 0;
      Z := X + Y;
   end Add;

begin
   null;
end Example_05;
```

This code is not correct. Postcondition is only relevant for normal returns.

1.12.6 Example #6

Listing 13: example_06.adb

```
procedure Example_06 is

   procedure Add (X, Y : Natural; Z : out Integer) with
     Post => Z = X + Y
   is
   begin
      Z := 0;
      Z := X + Y;
   end Add;
begin
   null;
end Example_06;
```

This code is correct. Procedure may raise an exception, but postcondition correctly describes normal returns.

1.12.7 Example #7

Listing 14: example_07.adb

```
procedure Example_07 is

   procedure Add (X, Y : Natural; Z : out Integer) with
     Pre  => X <= Integer'Last - Y,
     Post => Z = X + Y
   is
   begin
      Z := X + Y;
   end Add;
begin
   null;
end Example_07;
```

This code is correct. Precondition prevents exception inside Add. Postcondition is always satisfied.

1.12.8 Example #8

Listing 15: example_08.ads

```
package Example_08 is

   procedure Memset
     (B  : in out String;
      Ch :        Character;
      N  :        Natural)
     with
       Pre  => N <= B'Length,
       Post => (for all Idx in B'Range =>
                  (if Idx < B'First + N then
                      B (Idx) = Ch
                   else
                      B (Idx) = B (Idx)'Old));
end Example_08;
```

This code is not correct. `'Old` on expression including a quantified variable is not allowed.

1.12.9 Example #9

Listing 16: example_09.ads

```
package Example_09 is

   procedure Memset
     (B  : in out String;
      Ch :        Character;
      N  :        Natural)
     with
       Pre  => N <= B'Length - 1,
       Post => (for all Idx in 1 .. N => B (B'First + Idx - 1) = Ch)
                and then B (B'First + N) = B (B'First + N)'Old;

end Example_09;
```

This code is not correct. `Expr'Old` on potentially unevaluated expression is allowed only when `Expr` is a variable.

1.12.10 Example #10

Listing 17: example_10.ads

```
package Example_10 is

   procedure Memset
     (B  : in out String;
      Ch :        Character;
      N  :        Natural)
     with
       Pre  => N <= B'Length - 1,
```

(continues on next page)

(continued from previous page)

```
9         Post => (for all Idx in 1 .. N => B (B'First + Idx - 1) = Ch)
10                  and B (B'First + N) = B (B'First + N)'Old;
11
12  end Example_10;
```

This code is correct. `Expr'Old` does not appear anymore in a potentially unevaluated expression. Another solution would have been to apply `'Old` on B or to use **pragma** *Unevaluated_Use_Of_Old* (Allow).

CHAPTER

TWO

TYPE CONTRACTS

2.1 Type Contracts in Ada 2012 and SPARK 2014

- Natural evolution in Ada from previous type constraints
  - Scalar range specifies lower and upper bounds
  - Record discriminant specifies variants of the same type
- Executable type invariants by Meyer in Eiffel (1988)
  - Part of Design by Contract ™
  - Type invariant is checked dynamically when an object is created, and when an exported routine of the class returns
- Ada 2012 / SPARK 2014 support strong and weak invariants
  - A strong invariant must hold all the time
  - A weak invariant must hold outside of the scope of the type

2.2 Static and Dynamic Predicates

2.2.1 Static Predicate

- Original use case for type predicates in Ada 2012
  - Supporting non-contiguous subtypes of enumerations
  - Removes the constraint to define enumeration values in an order that allows defining interesting subtypes

Listing 1: show_static_predicate.ads

```
1  package Show_Static_Predicate is
2
3     type Day is (Monday,    Tuesday, Wednesday,
4                  Thursday, Friday,  Saturday,
5                  Sunday);
6
7     subtype Weekend is Day range Saturday .. Sunday;
8     subtype Day_Off is Day with
9       Static_Predicate => Day_Off in Wednesday | Weekend;
10
11 end Show_Static_Predicate;
```

- Typical use case on scalar types for holes in range

  - e.g. floats without `0.0`
- Types with static predicate are restricted
  - Cannot be used for the index of a loop or for array index (but OK for value tested in case statement)

2.2.2 Dynamic Predicate

- Extension of static predicate for any property
  - Property for static predicate must compare value to static expressions
  - Property for dynamic predicate can be anything

Listing 2: show_dynamic_predicate.ads

```
1  package Show_Dynamic_Predicate is
2
3     type Day is (Monday,   Tuesday, Wednesday,
4                  Thursday, Friday,  Saturday,
5                  Sunday);
6
7     function Check_Is_Off_In_Calendar (D : Day) return Boolean;
8
9     subtype Day_Off is Day with
10      Dynamic_Predicate => Check_Is_Off_In_Calendar (Day_Off);
11
12 end Show_Dynamic_Predicate;
```

- Various typical use cases on scalar and composite types
  - Strings that start at index 1 (`My_String'First = 1`)
  - Upper bound on record component that depends on the discriminant value (`Length <= Capacity`)
  - Ordering property on array values (`Is_Sorted (My_Array)`)

2.2.3 Restrictions on Types With Dynamic Predicate

- Types with dynamic predicate are restricted
  - Cannot be used for the index of a loop (same as static predicate)
  - Cannot be used as array index (same as static predicate)
  - Cannot be used for the value tested in a case statement
- No restriction on the property in Ada
  - Property can read the value of global variable (e.g. `Check_Is_Off_In_Calendar`)
    - what if global variable is updated?
  - Property can even have side-effects!
- Stronger restrictions on the property in SPARK
  - Property cannot read global variables or have side-effects
  - These restrictions make it possible to prove predicates

2.2.4 Dynamic Checking of Predicates

- Partly similar to other type constraints
  - Checked everywhere a range/discriminant check would be issued: assignment, parameter passing, type conversion, type qualification
  - ...but exception `Assertion_Error` is raised in case of violation
  - ...but predicates not checked by default, activated with `-gnata`
- Static predicate does not mean verification at compile time!

Listing 3: show_static_predicate_verified_at_runtime.ads

```
package Show_Static_Predicate_Verified_At_Runtime is

   type Day is (Monday,   Tuesday, Wednesday,
                Thursday, Friday,  Saturday,
                Sunday);

   subtype Weekend is Day range Saturday .. Sunday;
   subtype Day_Off is Day with
     Static_Predicate => Day_Off in Wednesday | Weekend;

   procedure Process_Day (This_Day : Day);

end Show_Static_Predicate_Verified_At_Runtime;
```

Listing 4: show_static_predicate_verified_at_runtime.adb

```
package body Show_Static_Predicate_Verified_At_Runtime is

   procedure Process_Day (This_Day : Day) is
      --  Predicate cannot be verified at compile time
      My_Day_Off : Day_Off := This_Day;
   begin
      --  missing implementation
      null;
   end Process_Day;

end Show_Static_Predicate_Verified_At_Runtime;
```

- Property should not contain calls to functions of the type
  - These functions will check the predicate on entry, leading to an infinite loop
  - GNAT compiler warns about such cases

2.2.5 Temporary Violations of the Dynamic Predicate

- Sometimes convenient to locally violate the property
  - Inside subprogram, to assign components of a record without an aggregate assignment
  - Violation even if no run-time check on component assignment
- Idiom is to define two types
  - First type does not have a predicate
  - Second type is a subtype of the first with the predicate
  - Conversions between these types at subprogram boundary

Listing 5: show_temp_violation_dyn_predicate.ads

```
package Show_Temp_Violation_Dyn_Predicate is

   type Day is (Monday,   Tuesday, Wednesday,
                Thursday, Friday,  Saturday,
                Sunday);

   type Raw_Week_Schedule is record
      Day_Off, Day_On_Duty : Day;
   end record;

   subtype Week_Schedule is Raw_Week_Schedule with
     Dynamic_Predicate =>
       Week_Schedule.Day_Off /= Week_Schedule.Day_On_Duty;

end Show_Temp_Violation_Dyn_Predicate;
```

2.3 Type Invariant

- Corresponds to the weak version of invariants
  - Predicates should hold always (only enforced with SPARK proof)
  - Type invariants should only hold outside of their defining package
- Type invariant can only be used on private types
  - Either on the private declaration
  - Or on the completion of the type in the private part of the package (makes more sense in general, only option in SPARK)

Listing 6: show_type_invariant.ads

```
package Show_Type_Invariant is

   type Day is (Monday,   Tuesday, Wednesday,
                Thursday, Friday,  Saturday,
                Sunday);

   type Week_Schedule is private;
private

   type Week_Schedule is record
      Day_Off, Day_On_Duty : Day;
   end record with
     Type_Invariant => Day_Off /= Day_On_Duty;

   procedure Internal_Adjust (WS : in out Week_Schedule);

end Show_Type_Invariant;
```

2.3.1 Dynamic Checking of Type Invariants

- Checked on outputs of public subprograms of the package
  - Checked on results of public functions
  - Checked on (**in**) **out** parameters of public subprograms
  - Checked on variables of the type, or having a part of the type
  - Exception `Assertion_Error` is raised in case of violation
  - Not checked by default, activated with `-gnata`
- No checking on internal subprograms!
  - Choice between predicate and type invariants depends on the need for such internal subprograms without checking

Listing 7: show_type_invariant.ads

```
package Show_Type_Invariant is

   type Day is (Monday,   Tuesday, Wednesday,
                Thursday, Friday,  Saturday,
                Sunday);

   type Week_Schedule is private;
private

   type Week_Schedule is record
      Day_Off, Day_On_Duty : Day;
   end record with
     Type_Invariant => Day_Off /= Day_On_Duty;

   procedure Internal_Adjust (WS : in out Week_Schedule);

end Show_Type_Invariant;
```

Listing 8: show_type_invariant.adb

```
package body Show_Type_Invariant is

   procedure Internal_Adjust (WS : in out Week_Schedule) is
   begin
      WS.Day_Off := WS.Day_On_Duty;
   end Internal_Adjust;

end Show_Type_Invariant;
```

2.4 Inheritance of Predicates and Type Invariants

- Derived types inherit the predicates of their parent type
  - Similar to other type constraints like bounds
  - Allows to structure a hierarchy of subtypes, from least to most constrained

Listing 9: show_predicate_inheritance.ads

```
package Show_Predicate_Inheritance is

   subtype String_Start_At_1 is String with
     Dynamic_Predicate => String_Start_At_1'First = 1;

   subtype String_Normalized is String_Start_At_1 with
     Dynamic_Predicate => String_Normalized'Last >= 0;

   subtype String_Not_Empty is String_Normalized with
     Dynamic_Predicate => String_Not_Empty'Length >= 1;

end Show_Predicate_Inheritance;
```

- Type invariants are typically not inherited
  - A private type cannot be derived unless it is tagged
  - Special aspect `Type_Invariant'Class` preferred for tagged types

2.5 Other Useful Gotchas on Predicates and Type Invariants

- GNAT defines its own aspects `Predicate` and `Invariant`
  - `Predicate` is the same as `Static_Predicate` if property allows it
  - Otherwise `Predicate` is the same as `Dynamic_Predicate`
  - `Invariant` is the same as `Type_Invariant`
- Referring to the *current object* in the property
  - The name of the type acts as the *current object* of that type
  - Components of records can be mentioned directly
- Type invariants on protected objects
  - Ada/SPARK do not define type invariants on protected objects

  - Idiom is to use a record type as unique component of the PO, and use a predicate for that record type

2.6 Default Initial Condition

- Aspect defined in GNAT to state a property on default initial values of a private type
  - Introduced for proof in SPARK
  - GNAT introduces a dynamic check when `-gnata` is used
  - Used in the formal containers library to state that containers are initially empty

Listing 10: show_default_init_cond.ads

```
with Ada.Containers;

package Show_Default_Init_Cond is

   type Count_Type is new Ada.Containers.Count_Type;

   type List (Capacity : Count_Type) is private with
      Default_Initial_Condition => Is_Empty (List);

   function Is_Empty (L : List) return Boolean;

private

   type List (Capacity : Count_Type) is null record;
   --  missing implementation...

end Show_Default_Init_Cond;
```

- Can also be used without a property for SPARK analysis
  - No argument specifies that the value is fully default initialized
  - Argument null specifies that there is no default initialization

2.7 Code Examples / Pitfalls

2.7.1 Example #1

Listing 11: example_01.ads

```
package Example_01 is

   type Day is (Monday,   Tuesday, Wednesday,
                Thursday, Friday,  Saturday,
                Sunday);

   subtype Weekend is Day range Saturday .. Sunday;

   subtype Day_Off is Day range Wednesday | Weekend;

end Example_01;
```

This code is not correct. The syntax of range constraints does not allow sets of values. A predicate should be used instead.

2.7.2 Example #2

Listing 12: example_02.ads

```
package Example_02 is

   type Day is (Monday,   Tuesday, Wednesday,
                Thursday, Friday,  Saturday,
                Sunday);

   subtype Weekend is Day range Saturday .. Sunday;

   subtype Day_Off is Weekend with
     Static_Predicate => Day_Off in Wednesday | Weekend;

end Example_02;
```

This code is not correct. This is accepted by GNAT, but result is not the one expected by the user. `Day_Off` has the same constraint as `Weekend`.

2.7.3 Example #3

Listing 13: example_03.ads

```
package Example_03 is

   type Day is (Monday,   Tuesday, Wednesday,
                Thursday, Friday,  Saturday,
                Sunday);

   subtype Weekend is Day range Saturday .. Sunday;

   subtype Day_Off is Day with
     Dynamic_Predicate => Day_Off in Wednesday | Weekend;

end Example_03;
```

This code is correct. It is valid to use a `Dynamic_Predicate` where a `Static_Predicate` would be allowed.

2.7.4 Example #4

Listing 14: week.ads

```
package Week is

   type Day is (Monday,   Tuesday, Wednesday,
                Thursday, Friday,  Saturday,
                Sunday);

   subtype Weekend is Day range Saturday .. Sunday;

   subtype Day_Off is Day with
```

(continues on next page)

(continued from previous page)

```
10       Static_Predicate => Day_Off in Wednesday | Weekend;
11
12    end Week;
```

Listing 15: example_04.adb

```
1  with Week; use Week;
2
3  procedure Example_04 is
4
5     function Next_Day_Off (D : Day_Off) return Day_Off is
6     begin
7        case D is
8           when Wednesday => return Saturday;
9           when Saturday  => return Sunday;
10          when Sunday    => return Wednesday;
11       end case;
12    end Next_Day_Off;
13
14 begin
15    null;
16 end Example_04;
```

This code is correct. It is valid to use a type with `Static_Predicate` for the value tested in a case statement. This is not true for `Dynamic_Predicate`.

2.7.5 Example #5

Listing 16: example_05.ads

```
1  package Example_05 is
2
3     type Day is (Monday,   Tuesday, Wednesday,
4                  Thursday, Friday,  Saturday,
5                  Sunday);
6
7     type Week_Schedule is private with
8       Type_Invariant => Valid (Week_Schedule);
9
10    function Valid (WS : Week_Schedule) return Boolean;
11
12 private
13    type Week_Schedule is record
14       Day_Off, Day_On_Duty : Day;
15    end record;
16
17    function Valid (WS : Week_Schedule) return Boolean is
18      (WS.Day_Off /= WS.Day_On_Duty);
19
20 end Example_05;
```

This code is correct. It is valid in Ada because the type invariant is not checked on entry or return from `Valid`. Also, function `Valid` is visible from the type invariant (special visibility in contracts). But it is invalid in SPARK, where private declaration cannot hold a type invariant. The reason is that the type invariant is assumed in the precondition of public functions for proof. That would lead to circular reasoning if `Valid` could be public.

2.7.6 Example #6

Listing 17: example_06.ads

```
package Example_06 is

   type Day is (Monday,   Tuesday, Wednesday,
                Thursday, Friday,  Saturday,
                Sunday);

   type Week_Schedule is private;

private

   type Week_Schedule is record
      Day_Off, Day_On_Duty : Day;
   end record with
     Type_Invariant => Valid (Week_Schedule);

   function Valid (WS : Week_Schedule) return Boolean is
     (WS.Day_Off /= WS.Day_On_Duty);

end Example_06;
```

This code is correct. This version is valid in both Ada and SPARK.

2.7.7 Example #7

Listing 18: example_07.ads

```
package Example_07 is

   subtype Sorted_String is String with
     Dynamic_Predicate =>
       (for all Pos in Sorted_String'Range =>
          Sorted_String (Pos) <= Sorted_String (Pos + 1));

   subtype Unique_String is String with
     Dynamic_Predicate =>
       (for all Pos1, Pos2 in Unique_String'Range =>
          Unique_String (Pos1) /= Unique_String (Pos2));

   subtype Unique_Sorted_String is String with
     Dynamic_Predicate =>
       Unique_Sorted_String in Sorted_String and then
       Unique_Sorted_String in Unique_String;

end Example_07;
```

This code is not correct. There are 3 problems in this code:

- there is a run-time error on the array access in `Sorted_String`;
- quantified expression defines only one variable;
- the property in `Unique_String` is true only for the empty string.

2.7.8 Example #8

Listing 19: example_08.ads

```
package Example_08 is

   subtype Sorted_String is String with
     Dynamic_Predicate =>
       (for all Pos in Sorted_String'First ..
          Sorted_String'Last - 1 =>
            Sorted_String (Pos) <= Sorted_String (Pos + 1));

   subtype Unique_String is String with
     Dynamic_Predicate =>
       (for all Pos1 in Unique_String'Range =>
          (for all Pos2 in Unique_String'Range =>
             (if Pos1 /= Pos2 then
                  Unique_String (Pos1) /= Unique_String (Pos2))));

   subtype Unique_Sorted_String is String with
     Dynamic_Predicate =>
       Unique_Sorted_String in Sorted_String and then
       Unique_Sorted_String in Unique_String;

end Example_08;
```

This code is correct. This is a correct version in Ada. For proving AoRTE in SPARK, one will need to change slightly the property of `Sorted_String`.

2.7.9 Example #9

Listing 20: example_09.ads

```
package Example_09 is

   type Day is (Monday,   Tuesday, Wednesday,
                Thursday, Friday,  Saturday,
                Sunday);

   type Week_Schedule is private with
     Default_Initial_Condition => Valid (Week_Schedule);

   function Valid (WS : Week_Schedule) return Boolean;

private

   type Week_Schedule is record
      Day_Off, Day_On_Duty : Day;
   end record;

   function Valid (WS : Week_Schedule) return Boolean is
     (WS.Day_Off /= WS.Day_On_Duty);

end Example_09;
```

This code is not correct. The default initial condition is not satisfied.

2.7.10 Example #10

Listing 21: example_10.ads

```
package Example_10 is

   type Day is (Monday,   Tuesday, Wednesday,
                Thursday, Friday,  Saturday,
                Sunday);

   type Week_Schedule is private with
     Default_Initial_Condition => Valid (Week_Schedule);

   function Valid (WS : Week_Schedule) return Boolean;

private

   type Week_Schedule is record
      Day_Off     : Day := Wednesday;
      Day_On_Duty : Day := Friday;
   end record;

   function Valid (WS : Week_Schedule) return Boolean is
     (WS.Day_Off /= WS.Day_On_Duty);

end Example_10;
```

This code is correct. This is a correct version, which can be proved with SPARK.

CHAPTER

THREE

SYSTEMS PROGRAMMING

3.1 Type Contracts in Ada 2012 and SPARK 2014

3.2 Systems Programming – What is it?

- Bare metal programming
  - bare board applications (no Operating System)
  - Operating Systems (ex: Muen separation kernel)
  - device drivers (ex: Ada Drivers Library)
  - communication stacks (ex: AdaCore TCP/IP stack)
- Specifics of Systems Programming
  - direct access to hardware: registers, memory, etc.
  - side-effects (yes!)
  - efficiency is paramount (sometimes real-time even)
  - hard/impossible to debug

3.3 Systems Programming – How can SPARK help?

- SPARK is a Systems Programming language
  - same features as Ada for accessing hardware (representation clauses, address clauses)
  - as efficient as Ada or C
- Side-effects can be modeled in SPARK
  - reads and writes to memory-mapped devices are modeled
  - concurrent interactions with environment are modeled
- SPARK can help catch problems by static analysis
  - correct flows, initialization, concurrent accesses
  - absence of run-time errors and preservation of invariants

3.4 Systems Programming – A trivial example

Listing 1: show_trivial_sys_prog.ads

```
package Show_Trivial_Sys_Prog is

   Y : Integer;

   --  Y'Address could be replaced by any
   --  external address
   X : Integer with Volatile,
     Address => Y'Address;

   procedure Get (Val : out Integer)
     with Global  => (In_Out => X),
     Depends => (Val => X,
                 X   => X);

end Show_Trivial_Sys_Prog;
```

Listing 2: show_trivial_sys_prog.adb

```
package body Show_Trivial_Sys_Prog is

   procedure Get (Val : out Integer) is
   begin
      Val := X;
   end Get;

end Show_Trivial_Sys_Prog;
```

- Comments:
  - X is volatile
  - X is also an output; output X depends on input X
  - X is only read

3.5 Volatile Variables and Volatile Types

- Variables whose reads/writes cannot be optimized away
- Identified through multiple aspects (or pragmas)
  - aspect `Volatile`
  - but also aspect `Atomic`
  - and GNAT aspect `Volatile_Full_Access`
  - all the above aspects can be set on type or object
- Other aspects are useful on volatile variables
  - aspect **`Address`** to specify location in memory
  - aspect `Import` to skip definition/initialization

```
type T is new Integer with Volatile;

X : Integer with Atomic, Import, Address => ... ;
```

3.6 Flavors of Volatile Variables

3.6.1 Using Async_Readers / Async_Writers

- Boolean aspects describing asynchronous behavior
  - Async_Readers if variable may be read asynchronously
  - Async_Writers if variable may be written asynchronously
- Effect of Async_Readers on flow analysis
- Effect of Async_Writers on flow analysis & proof
  - always initialized, always has an unknown value

Listing 3: volatile_vars.ads

```
1  package Volatile_Vars is
2
3     pragma Elaborate_Body;
4
5     Ext : array (1 .. 2) of Integer;
6
7     X : Integer with Volatile,
8       Address => Ext (1)'Address,
9       Async_Readers;
10
11    Y : Integer with Volatile,
12      Address => Ext (2)'Address,
13      Async_Writers;
14
15    procedure Set;
16 end Volatile_Vars;
```

Listing 4: volatile_vars.adb

```
1  package body Volatile_Vars is
2
3     procedure Set is
4        U, V : constant Integer := Y;
5     begin
6        pragma Assert (U = V);
7        X := 0;
8        X := 1;
9     end Set;
10 begin
11    Ext := (others => 0);
12 end Volatile_Vars;
```

Listing 5: show_volatile_vars.adb

```
with Volatile_Vars;

procedure Show_Volatile_Vars is
begin
   Volatile_Vars.Set;
end Show_Volatile_Vars;
```

3.6.2 Using Effective_Reads / Effective_Writes

- Boolean aspects distinguishing values & sequences
  - `Effective_Reads` if reading the variable has an effect on its value
  - `Effective_Writes` if writing the variable has an effect on its value
- Effect of both on proof and flow dependencies
  - Final value of variable is seen as a sequence of values it took

Listing 6: volatile_vars.ads

```
package Volatile_Vars is

   pragma Elaborate_Body;

   Ext : array (1 .. 2) of Integer;

   X : Integer with Volatile,
     Address => Ext (1)'Address,
     Async_Readers,
     Effective_Writes;

   Y : Integer with Volatile,
     Address => Ext (2)'Address,
     Async_Writers,
     Effective_Reads;

   procedure Set with
      Depends => (X => Y,
                  Y => Y);
end Volatile_Vars;
```

Listing 7: volatile_vars.adb

```
package body Volatile_Vars is

   procedure Set is
   begin
      X := Y;
      X := 0;
   end Set;

begin
   Ext := (others => 0);
end Volatile_Vars;
```

Listing 8: show_volatile_vars.adb

```
with Volatile_Vars;

procedure Show_Volatile_Vars is
begin
   Volatile_Vars.Set;
end Show_Volatile_Vars;
```

3.6.3 Combinations of Flavors of Volatile Variables

- All four flavors can be set independently
  - Default for Volatile/Atomic is all four `True`
  - When some aspects set, all others default to `False`
- Only half the possible combinations are legal
  - `Async_Readers` and/or `Async_Writers` is set
  - `Effective_Reads = True` forces `Async_Writers = True`
  - `Effective_Writes = True` forces `Async_Readers = True`
  - sensor: AW=`True`
  - actuator: AR=`True`
  - input port: AW=`True`, ER=`True`
  - output port: AR=`True`, EW=`True`

3.7 Constraints on Volatile Variables

- Volatile variables must be defined at library level
- Expressions (and functions) cannot have side-effects
  - read of variable with AW=`True` must appear alone on *rhs* of assign
  - a function cannot read a variable with ER=`True`

Listing 9: volatile_vars.ads

```
package Volatile_Vars is

   pragma Elaborate_Body;

   Ext : array (1 .. 4) of Integer;

   AR : Integer with Volatile,
     Address => Ext (1)'Address,
     Async_Readers;

   AW : Integer with Volatile,
     Address => Ext (2)'Address,
     Async_Writers;

   ER : Integer with Volatile,
     Address => Ext (3)'Address,
```

(continues on next page)

(continued from previous page)

```
     Async_Writers,
     Effective_Reads;

   EW : Integer with Volatile,
     Address => Ext (4)'Address,
     Async_Readers,
     Effective_Writes;

   procedure Read_All;

   function Read_ER return Integer;

   procedure Set (V : Integer);

end Volatile_Vars;
```

Listing 10: volatile_vars.adb

```
package body Volatile_Vars is

   procedure Read_All is
      Tmp : Integer := 0;
   begin
      Tmp := Tmp + AR;
      Tmp := Tmp + AW;
      EW := Tmp;
      Set (ER);
   end Read_All;

   function Read_ER return Integer is
      Tmp : Integer := ER;
   begin
      return Tmp;
   end Read_ER;

   procedure Set (V : Integer) is
   begin
      AW := V;
   end Set;

begin
   Ext := (others => 0);
end Volatile_Vars;
```

Listing 11: show_volatile_vars.adb

```
with Volatile_Vars;

procedure Show_Volatile_Vars is
   V : Integer;
begin
   Volatile_Vars.Read_All;
   V := Volatile_Vars.Read_ER;
end Show_Volatile_Vars;
```

- Comments:
  - AW not alone on rhs
  - ER not alone on rhs
  - ER output of Read_ER

3.8 Constraints on Volatile Functions

- Functions should have mathematical interpretation
  - a function reading a variable with AW=**True** is marked as volatile with aspect `Volatile_Function`
  - calls to volatile functions are restricted like reads of `Async_Writers`

Listing 12: volatile_vars.ads

```
package Volatile_Vars is

   pragma Elaborate_Body;

   Ext : array (1 .. 4) of Integer;

   AR : Integer with Volatile,
     Address => Ext (1)'Address,
     Async_Readers;

   AW : Integer with Volatile,
     Address => Ext (2)'Address,
     Async_Writers;

   ER : Integer with Volatile,
     Address => Ext (3)'Address,
     Async_Writers,
     Effective_Reads;

   EW : Integer with Volatile,
     Address => Ext (4)'Address,
     Async_Readers,
     Effective_Writes;

   function Read_Non_Volatile
     return Integer;

   function Read_Volatile
     return Integer
     with Volatile_Function;

   function Read_ER
     return Integer
     with Volatile_Function;

end Volatile_Vars;
```

Listing 13: volatile_vars.adb

```
package body Volatile_Vars is

   function Read_Non_Volatile
     return Integer is
      Tmp : Integer := 0;
   begin
      --  reads AR, AW, EW
      --  ERROR: not a volatile function
      Tmp := Tmp + AR;
      Tmp := Tmp + AW;
      Tmp := Tmp + EW;

```

(continues on next page)

(continued from previous page)

```
      return Tmp;
   end Read_Non_Volatile;

   function Read_Volatile
     return Integer is
      Tmp : Integer := 0;
   begin
      --  reads AR, AW, EW
      --  OK for volatile function
      Tmp := Tmp + AR;
      Tmp := Tmp + AW;
      Tmp := Tmp + EW;

      return Tmp;
   end Read_Volatile;

   function Read_ER
     return Integer is
      Tmp : Integer := ER;
   begin
      --  reads ER
      --  ERROR: ER output of Read_ER
      return Tmp;
   end Read_ER;

begin
   Ext := (others => 0);
end Volatile_Vars;
```

Listing 14: show_volatile_vars.adb

```
with Volatile_Vars;

procedure Show_Volatile_Vars is
   V : Integer;
begin
   V := Volatile_Vars.Read_Non_Volatile;
   V := Volatile_Vars.Read_Volatile;
   V := Volatile_Vars.Read_ER;
end Show_Volatile_Vars;
```

3.9 State Abstraction on Volatile Variables

- Abstract state needs to be identified as `External`
- Flavors of volatility can be specified
  - Default if none specified is all True

Listing 15: p1.ads

```
package P1 with
  Abstract_State => (S with External)
is
   procedure Process (Data : out Integer) with
     Global => (In_Out => S);

end P1;
```

Listing 16: p2.ads

```
package P2 with
  Abstract_State => (S with External =>
                       (Async_Writers,
                        --  OK if refined into AW, ER
                        Effective_Reads)
                        --  not OK if refined into AR, EW
                     )
is
   procedure Process (Data : out Integer) with
     Global => (In_Out => S);

end P2;
```

3.10 Constraints on Address Attribute

- Address of volatile variable can be specified

Listing 17: show_address_attribute.ads

```
package Show_Address_Attribute is

   Ext : array (1 .. 2) of Integer;

   X : Integer with Volatile,
     Address => Ext (1)'Address;

   Y : Integer with Volatile;
   for Y'Address use Ext (2)'Address;

end Show_Address_Attribute;
```

- Address attribute not allowed in expressions
- Overlays are allowed
  - GNATprove does not check absence of overlays
  - GNATprove does not model the resulting aliasing

Listing 18: show_address_overlay.adb

```
procedure Show_Address_Overlay is

   X : Integer := 1;
   Y : Integer := 0
     with Address => X'Address;

   pragma Assert (X = 1);
   --  assertion wrongly proved
begin
   null;

end Show_Address_Overlay;
```

3.11 Can something be known of volatile variables?

- Variables with `Async_Writers` have no known value
- ... but they have a known type!
  - type range, ex: `0 .. 360`
  - type predicate, ex: `0 .. 15 | 17 .. 42 | 43 .. 360`
- Variables without `Async_Writers` have a known value
- GNATprove also assumes all values are valid (`X'Valid`)

Listing 19: show_provable_volatile_var.ads

```
package Show_Provable_Volatile_Var is

   X : Integer with Volatile, Async_Readers;

   procedure Read_Value;

end Show_Provable_Volatile_Var;
```

Listing 20: show_provable_volatile_var.adb

```
package body Show_Provable_Volatile_Var is

   procedure Read_Value is
   begin
      X := 42;
      pragma Assert (X = 42);
      --  proved!
   end Read_Value;

end Show_Provable_Volatile_Var;
```

3.12 Other Concerns in Systems Programming

- Software startup state ⟶ elaboration rules
  - SPARK follows Ada static elaboration model
  - ... with additional constraints for ensuring correct initialization
  - ... but GNATprove follows the relaxed GNAT static elaboration
- Handling of faults ⟶ exception handling
  - raising exceptions is allowed in SPARK
  - ... but exception handlers are `SPARK_Mode => Off`
  - ... typically the last-chance-handler is used instead
- Concurrency inside the application ⟶ tasking support
  - Ravenscar and Extended_Ravenscar profiles supported in SPARK

3.13 Code Examples / Pitfalls

3.13.1 Example #1

Listing 21: example_01.ads

```
package Example_01 is

   Ext : Integer;

   X   : Integer with Volatile,
     Address => Ext'Address;

   procedure Get (Val : out Integer)
     with Global  => (Input => X),
     Depends => (Val => X);

end Example_01;
```

Listing 22: example_01.adb

```
package body Example_01 is

   procedure Get (Val : out Integer) is
   begin
      Val := X;
   end Get;

end Example_01;
```

This code is not correct. X has `Effective_Reads` set by default, hence it is also an output.

3.13.2 Example #2

Listing 23: example_02.ads

```
package Example_02 is

   Ext : Integer;

   X : Integer with Volatile, Address => Ext'Address,
     Async_Readers, Async_Writers, Effective_Writes;

   procedure Get (Val : out Integer)
     with Global  => (Input => X),
     Depends => (Val => X);

end Example_02;
```

Listing 24: example_02.adb

```
package body Example_02 is

   procedure Get (Val : out Integer) is
   begin
      Val := X;
   end Get;

end Example_02;
```

This code is correct. X has `Effective_Reads = False`, hence it is only an input.

3.13.3 Example #3

Listing 25: example_03.ads

```
package Example_03 is

   Speed : Float with Volatile, Async_Writers;
   Motor : Float with Volatile, Async_Readers;

   procedure Adjust with
     Depends => (Motor =>+ Speed);

end Example_03;
```

Listing 26: example_03.adb

```
package body Example_03 is

   procedure Adjust is
      Cur_Speed : constant Float := Speed;
   begin
      if abs (Cur_Speed) > 100.0 then
         Motor := Motor - 1.0;
      end if;
   end Adjust;

end Example_03;
```

This code is correct. Speed is an input only, `Motor` is both an input and output. Note how the current value of Speed is first copied to be tested in a larger expression.

3.13.4 Example #4

Listing 27: example_04.ads

```
package Example_04 is

   Raw_Data : Float with Volatile,
     Async_Writers, Effective_Reads;
   Data     : Float with Volatile,
     Async_Readers, Effective_Writes;

   procedure Smooth with
     Depends => (Data => Raw_Data);

end Example_04;
```

Listing 28: example_04.adb

```
package body Example_04 is

   procedure Smooth is
      Data1 : constant Float := Raw_Data;
      Data2 : constant Float := Raw_Data;
   begin
      Data := Data1;
      Data := (Data1 + Data2) / 2.0;
      Data := Data2;
   end Smooth;

end Example_04;
```

This code is not correct. Raw_Data has `Effective_Reads` set, hence it is also an output.

3.13.5 Example #5

Listing 29: example_05.ads

```
package Example_05 is

   type Regval is new Integer with Volatile;
   type Regnum is range 1 .. 32;
   type Registers is array (Regnum) of Regval;

   Regs : Registers with Async_Writers, Async_Readers;

   function Reg (R : Regnum) return Integer is
     (Integer (Regs (R))) with Volatile_Function;

end Example_05;
```

This code is not correct. Regs has Async_Writers set, hence it cannot appear as the expression in an expression function.

3.13.6 Example #6

Listing 30: example_06.ads

```
package Example_06 is

   type Regval is new Integer with Volatile;
   type Regnum is range 1 .. 32;
   type Registers is array (Regnum) of Regval;

   Regs : Registers with Async_Writers, Async_Readers;

   function Reg (R : Regnum) return Integer
     with Volatile_Function;

end Example_06;
```

Listing 31: example_06.adb

```
package body Example_06 is

   function Reg (R : Regnum) return Integer is
      V : Regval := Regs (R);
   begin
      return Integer (V);
   end Reg;

end Example_06;
```

This code is not correct. Regval is a volatile type, hence variable V is volatile and cannot be declared locally.

3.13.7 Example #7

Listing 32: example_07.ads

```
package Example_07 is

   type Regval is new Integer with Volatile;
   type Regnum is range 1 .. 32;
   type Registers is array (Regnum) of Regval;

   Regs : Registers with Async_Writers, Async_Readers;

   function Reg (R : Regnum) return Integer
     with Volatile_Function;

end Example_07;
```

Listing 33: example_07.adb

```
package body Example_07 is

   function Reg (R : Regnum) return Integer is
   begin
      return Integer (Regs (R));
   end Reg;

end Example_07;
```

This code is correct. Regs has `Effective_Reads = False` hence can be read in a function. Function Reg is marked as volatile with aspect `Volatile_Function`. No volatile variable is declared locally.

3.13.8 Example #8

Listing 34: example_08.ads

```
package Example_08 with
  Abstract_State => (State with External),
    Initializes => State
is
   procedure Dummy;
end Example_08;
```

Listing 35: example_08.adb

```
package body Example_08 with
  Refined_State => (State => (X, Y, Z))
is
   X : Integer with Volatile, Async_Readers;
   Y : Integer with Volatile, Async_Writers;
   Z : Integer := 0;

   procedure Dummy is
   begin
      null;
   end Dummy;

end Example_08;
```

This code is not correct. X has `Async_Writers = False`, hence is not considered as always initialized. As aspect `Initializes` specifies that `State` should be initialized after elaboration, this is an error. Note that is allowed to bundle volatile and non-volatile variables in an external abstract state.

3.13.9 Example #9

Listing 36: example_09.ads

```
package Example_09 is

   type Pair is record
      U, V : Natural;
   end record
     with Predicate => U /= V;

   X : Pair with Atomic, Async_Readers, Async_Writers;

   function Max return Integer with
     Volatile_Function,
     Post => Max'Result /= 0;

end Example_09;
```

Listing 37: example_09.adb

```
package body Example_09 is

   function Max return Integer is
      Val1 : constant Natural := X.U;
      Val2 : constant Natural := X.V;
   begin
```

(continues on next page)

(continued from previous page)

```
      return Natural'Max (Val1, Val2);
   end Max;

end Example_09;
```

This code is not correct. `X` has `Async_Writers` set, hence it may have been written between the successive reads of `X.U` and `X.V`.

3.13.10 Example #10

Listing 38: example_10.ads

```
package Example_10 is

   type Pair is record
      U, V : Natural;
   end record
     with Predicate => U /= V;

   X : Pair with Atomic, Async_Readers, Async_Writers;

   function Max return Integer with
     Volatile_Function,
     Post => Max'Result /= 0;

end Example_10;
```

Listing 39: example_10.adb

```
package body Example_10 is

   function Max return Integer is
      P    : constant Pair := X;
      Val1 : constant Natural := P.U;
      Val2 : constant Natural := P.V;
   begin
      return Natural'Max (Val1, Val2);
   end Max;

end Example_10;
```

This code is correct. Values of `P.U` and `P.V` are provably different, and the postcondition is proved.

CHAPTER

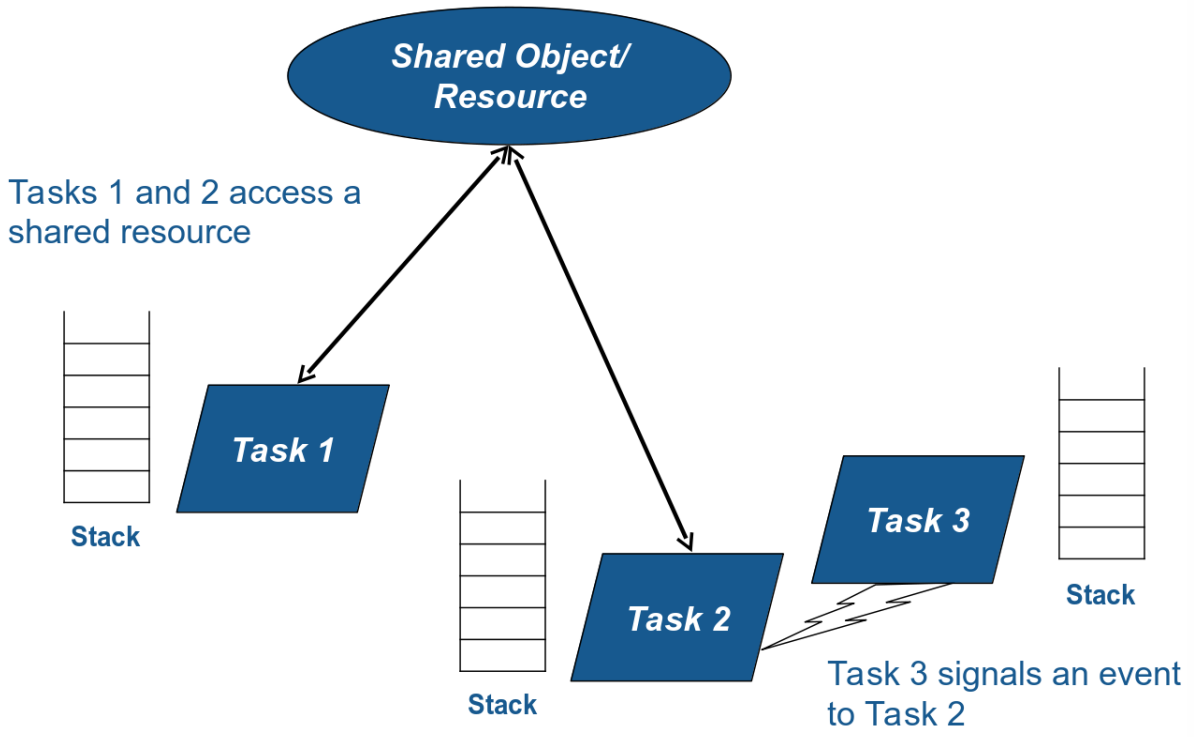
FOUR

CONCURRENCY

4.1 Concurrency ≠ Parallelism

- Concurrency allows to create a well structured program
- Parallelism allows to create a high performance program
- Multiple cores/processors are...
  - possible for concurrent programs
  - essential to parallelism
- What about Ada and SPARK?
  - GNAT runtimes for concurrency available on single core & multicore (for SMP platforms)
  - parallel features scheduled for inclusion in Ada and SPARK 202x

4.2 Concurrent Program Structure in Ada



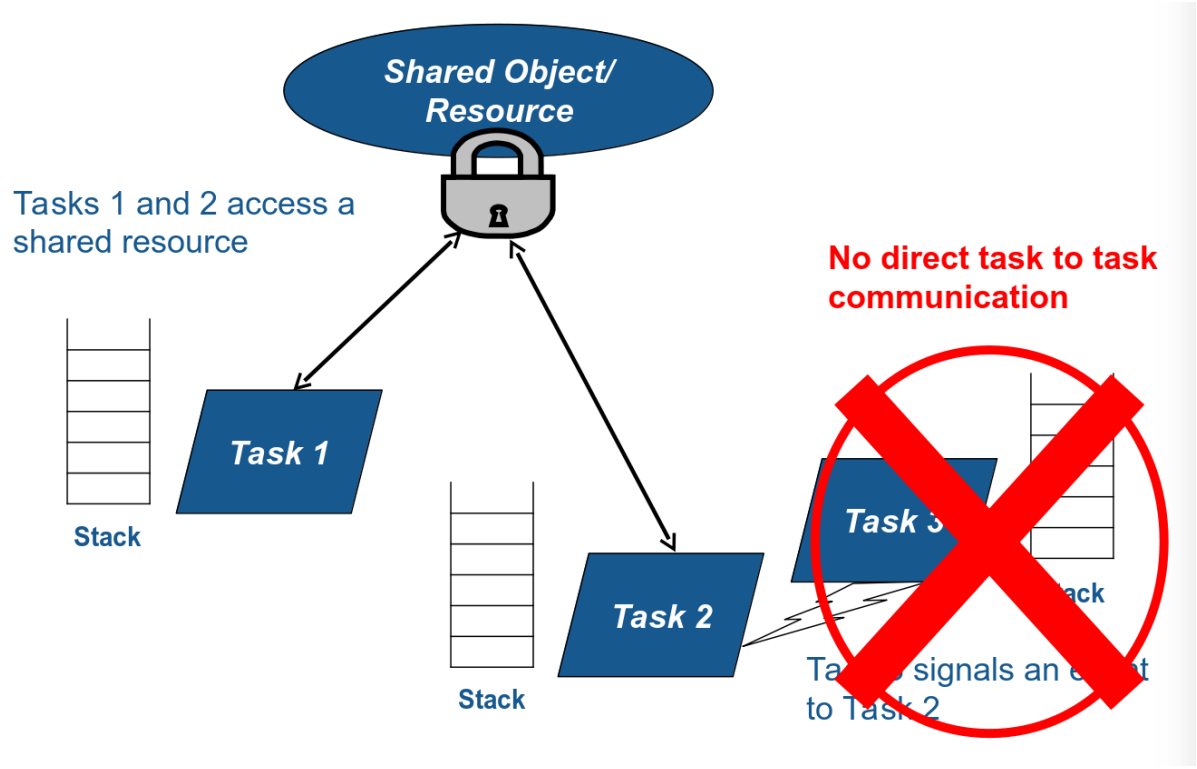
4.3 The problems with concurrency

- Control and data flow become much more complex
  - possibly nondeterministic even
  - actual behavior is one of many possible interleavings of tasks
- Data may be corrupted by concurrent accesses
  - so called data races or race conditions
- Control may block indefinitely, or loop indefinitely
  - so called deadlocks and livelocks
- Scheduling and memory usage are harder to compute

4.4 Ravenscar – the Ada solution to concurrency problems

- Ravenscar profile restricts concurrency in Ada
  - ensures deterministic behavior at every point in time
  - recommends use of protected objects to avoid data races
  - prevents deadlocks with Priority Ceiling Protocol
  - allows use of scheduling analysis techniques (RMA, RTA)
  - facilitates computation of memory usage with static tasks
- GNAT Extended Ravenscar profile lifts some restrictions
  - still same benefits as Ravenscar profile
  - removes painful restrictions for some applications

4.5 Concurrent Program Structure in Ravenscar



4.6 Ravenscar – the SPARK solution to concurrency problems

- Ravenscar and Extended_Ravenscar profiles supported in SPARK
- Data races prevented by flow analysis
  - ensures no problematic concurrent access to unprotected data
  - flow analysis also ensures non-termination of tasks
- Run-time errors prevented by proof
  - includes violations of the Priority Ceiling Protocol

4.7 Concurrency – A trivial example

Listing 1: show_trivial_task.ads

```
1 package Show_Trivial_Task is
2
3    type Task_Id is new Integer;
4
5    task type T (Id : Task_Id);
6
7    T1 : T (0);
8    T2 : T (1);
9 end Show_Trivial_Task;
```

Listing 2: show_trivial_task.adb

```
package body Show_Trivial_Task is
   task body T is
      Current_Task : Task_Id := Id;
   begin
      loop
         delay 1.0;
      end loop;
   end T;
end Show_Trivial_Task;
```

- Id can be written by T1 and T2 at the same time

4.8 Setup for using concurrency in SPARK

- Any unit using concurrency features (tasks, protected objects, etc.) must set the profile

```
pragma Profile (Ravenscar);
--  or
pragma Profile (GNAT_Extended_Ravenscar);
```

- ... plus an additional pragma
  - that ensures tasks start after the end of elaboration

```
pragma Partition_Elaboration_Policy (Sequential);
```

- ... which are checked by GNAT partition-wide
  - pragmas needed for verification even it not for compilation

4.9 Tasks in Ravenscar

- A task can be either a singleton object or a type
  - no declarations of entries for rendez-vous

```
task T;
task type TT;
```

- ... completed by a body
  - infinite loop to prevent termination

```
task body T is
begin
   loop
      ...
   end loop;
end T;
```

- Tasks are declared at library-level
- ... as standalone objects or inside records/arrays

```
type TA is array (1 .. 3) of TT;
type TR is record
   A, B : TT;
end record;
```

4.10 Communication Between Tasks in Ravenscar

- Tasks can communicate through protected objects
- A protected object is either a singleton object or a type
  - all PO private data initialized by default in SPARK

Listing 3: show_protected_object.ads

```
1  package Show_Protected_Object is
2
3     protected P is
4        procedure Set (V : Natural);
5        function Get return Natural;
6     private
7        The_Data : Natural := 0;
8     end P;
9
10 end Show_Protected_Object;
```

Listing 4: show_protected_object.adb

```
1  package body Show_Protected_Object is
2
3     protected body P is
4        procedure Set (V : Natural) is
5        begin
6           The_Data := V;
7        end Set;
8        function Get return Natural is
9          (The_Data);
10    end P;
11
12 end Show_Protected_Object;
```

4.11 Protected Objects in Ravenscar

- Protected objects are declared at library-level
- ... as standalone objects or inside records/arrays
  - The record type needs to be volatile, as a non-volatile type cannot contain a volatile component. The array type is implicitly volatile when its component type is volatile.

Listing 5: show_protected_object_ravenscar.ads

```
1  package Show_Protected_Object_Ravenscar is
2
3     protected type PT is
```

(continues on next page)

(continued from previous page)

```
      procedure Set (V : Natural);
      function Get return Natural;
   private
      The_Data : Natural := 0;
   end PT;

   P : PT;

   type PAT is array (1 .. 3) of PT;
   PA : PAT;

   type PRT is record
      A, B : PT;
   end record with Volatile;
   PR : PRT;

end Show_Protected_Object_Ravenscar;
```

Listing 6: show_protected_object_ravenscar.adb

```
package body Show_Protected_Object_Ravenscar is

   protected body PT is
      procedure Set (V : Natural) is
      begin
         The_Data := V;
      end Set;
      function Get return Natural is
        (The_Data);
   end PT;

end Show_Protected_Object_Ravenscar;
```

4.12 Protected Communication with Procedures & Functions

- CREW enforced (Concurrent-Read-Exclusive-Write)
  - procedures have exclusive read-write access to PO
  - functions have shared read-only access to PO
- Actual mechanism depends on target platform
  - scheduler enforces policy on single core
  - locks used on multicore (using CAS instructions)
  - lock-free transactions used for simple PO (again using CAS)
- Mechanism is transparent to user
  - user code simply calls procedures/functions
  - task may be queued until PO is released by another task

4.13 Blocking Communication with Entries

- Only protected objects have entries in Ravenscar
- Entry = procedure with **entry** guard condition
  - second level of queues, one for each entry, on a given PO
  - task may be queued until guard is True and PO is released
  - at most one entry in Ravenscar
  - guard is a **Boolean** component of PO in Ravenscar

Listing 7: show_blocking_communication.ads

```
package Show_Blocking_Communication is

   protected type PT is
      entry Reset;
   private
      Is_Not_Null : Boolean := False;
      The_Data    : Integer := 1000;
   end PT;

end Show_Blocking_Communication;
```

Listing 8: show_blocking_communication.adb

```
package body Show_Blocking_Communication is

   protected body PT is
      entry Reset when Is_Not_Null is
      begin
         The_Data := 0;
      end Reset;
   end PT;

end Show_Blocking_Communication;
```

4.14 Relaxed Constraints on Entries with Extended Ravenscar

- Proof limitations with Ravenscar
  - not possible to relate guard to other components with invariant
- GNAT Extended Ravenscar profile lifts these constraints
  - and allows multiple tasks to call the same entry

Listing 9: show_relaxed_constraints_on_entries.ads

```
package Show_Relaxed_Constraints_On_Entries is

   protected type Mailbox is
      entry Publish;
      entry Retrieve;
   private
      Num_Messages : Natural := 0;
```

(continues on next page)

(continued from previous page)

```
   end Mailbox;

end Show_Relaxed_Constraints_On_Entries;
```

Listing 10: show_relaxed_constraints_on_entries.adb

```
package body Show_Relaxed_Constraints_On_Entries is

   Max : constant := 100;

   protected body Mailbox is
      entry Publish when Num_Messages < Max is
      begin
         Num_Messages := Num_Messages + 1;
      end Publish;

      entry Retrieve when Num_Messages > 0 is
      begin
         Num_Messages := Num_Messages - 1;
      end Retrieve;
   end Mailbox;

end Show_Relaxed_Constraints_On_Entries;
```

4.15 Interrupt Handlers in Ravenscar

- Interrupt handlers are parameterless procedures of PO
  - with aspect `Attach_Handler` specifying the corresponding signal
  - with aspect `Interrupt_Priority` on the PO specifying the priority

Listing 11: show_interrupt_handlers.ads

```
with System; use System;
with Ada.Interrupts.Names; use Ada.Interrupts.Names;

package Show_Interrupt_Handlers is

    protected P with
      Interrupt_Priority =>
        System.Interrupt_Priority'First
    is
       procedure Signal with
         Attach_Handler => SIGHUP;
    end P;

end Show_Interrupt_Handlers;
```

- Priority of the PO should be in `System.Interrupt_Priority`
  - default is OK – in the range of `System.Interrupt_Priority`
  - checked by proof (default or value of `Priority` or `Interrupt_Priority`)

4.16 Other Communications Between Tasks in SPARK

- Tasks must communicate through synchronized objects
  - atomic objects
  - protected objects
  - suspension objects (standard `Boolean` protected objects)
- Constants are considered as synchronized
  - this includes variables constant after elaboration (specified with aspect `Constant_After_Elaboration`)
- Single task or PO can access an unsynchronized object
  - exclusive relation between object and task/PO must be specified with aspect `Part_Of`

4.17 Data and Flow Dependencies of Tasks

- Input/output relation can be specified for a task
  - as task never terminates, output is understood while task runs
  - task itself is both an input and an output
  - implicit `In_Out => T`
  - explicit dependency

Listing 12: show_data_and_flow_dependencies.ads

```
1  package Show_Data_And_Flow_Dependencies is
2
3     X, Y, Z : Integer;
4
5     task T with
6       Global => (Input  => X,
7                  Output => Y,
8                  In_Out => Z),
9       Depends => (T    => T,
10                  Z    => X,
11                  Y    => X,
12                  null => Z);
13 end Show_Data_And_Flow_Dependencies;
```

4.18 State Abstraction over Synchronized Variables

- Synchronized objects can be abstracted in synchronized abstract state with aspect `Synchronous`

Listing 13: show_state_abstraction.ads

```
1  package Show_State_Abstraction with
2    Abstract_State => (State with Synchronous, External)
3  is
4
```

(continues on next page)

(continued from previous page)

```
   protected type Protected_Type is
      procedure Reset;
   private
      Data : Natural := 0;
   end Protected_Type;

   task type Task_Type;

end Show_State_Abstraction;
```

Listing 14: show_state_abstraction.adb

```
package body Show_State_Abstraction with
  Refined_State => (State => (A, P, T))
is
   A : Integer with Atomic, Async_Readers, Async_Writers;
   P : Protected_Type;
   T : Task_Type;

   protected body Protected_Type is
      procedure Reset is
      begin
         Data := 0;
      end Reset;
   end Protected_Type;

   task body Task_Type is
   begin
      P.Reset;
      A := 0;
   end Task_Type;

end Show_State_Abstraction;
```

- Synchronized state is a form of external state
  - Synchronous same as `External => (Async_Readers, Async_Writers)`
  - tasks are not volatile and can be part of regular abstract state

4.19 Synchronized Abstract State in the Standard Library

- Standard library maintains synchronized state
  - the tasking runtime maintains state about running tasks
  - the real-time runtime maintains state about current time

```
package Ada.Task_Identification with
  SPARK_Mode,
  Abstract_State =>
    (Tasking_State with Synchronous,
       External => (Async_Readers, Async_Writers)),
  Initializes    => Tasking_State

package Ada.Real_Time with
  SPARK_Mode,
  Abstract_State =>
```

(continues on next page)

(continued from previous page)

```
    (Clock_Time with Synchronous,
       External => (Async_Readers, Async_Writers)),
  Initializes    => Clock_Time
```

- API of these units refer to `Tasking_State` and `Clock_Time`

4.20 Code Examples / Pitfalls

4.20.1 Example #1

Listing 15: rendezvous.adb

```
procedure Rendezvous is
   task T1 is
      entry Start;
   end T1;

   task body T1 is
   begin
      accept Start;
   end T1;

begin
   T1.Start;
end Rendezvous;
```

This code is not correct. Task rendezvous is not allowed; violation of restriction `Max_Task_Entries = 0`. A local task is not allowed; violation of restriction `No_Task_Hierarchy`

4.20.2 Example #2

Listing 16: example_02.ads

```
package Example_02 is

   protected P is
      entry Reset;
   end P;

private
   Data : Boolean := False;
end Example_02;
```

Listing 17: example_02.adb

```
package body Example_02 is

   protected body P is
      entry Reset when Data is
      begin
         null;
      end Reset;
   end P;
```

(continues on next page)

(continued from previous page)

```
9
10 end Example_02;
```

This code is not correct. Global data in entry guard is not allowed. Violation of restriction `Simple_Barriers` (for Ravenscar) or `Pure_Barriers` (for Extended Ravenscar)

4.20.3 Example #3

Listing 18: example_03.ads

```
1  package Example_03 is
2
3     protected P is
4        procedure Set (Value : Integer);
5     end P;
6
7  private
8     task type TT;
9
10    T1, T2 : TT;
11
12 end Example_03;
```

Listing 19: example_03.adb

```
1  package body Example_03 is
2
3     Data : Integer := 0;
4
5     protected body P is
6        procedure Set (Value : Integer) is
7        begin
8           Data := Value;
9        end Set;
10    end P;
11
12    task body TT is
13       Local : Integer := 0;
14    begin
15       loop
16          Local := (Local + 1) mod 100;
17          P.Set (Local);
18       end loop;
19    end TT;
20
21 end Example_03;
```

This code is not correct. Global unprotected data accessed in protected object shared between tasks

4.20.4 Example #4

Listing 20: example_04.ads

```
package Example_04 is

   protected P is
      procedure Set (Value : Integer);
   end P;

private
   Data : Integer := 0 with Part_Of => P;

   task type TT;

   T1, T2 : TT;

end Example_04;
```

Listing 21: example_04.adb

```
package body Example_04 is

   protected body P is
      procedure Set (Value : Integer) is
      begin
         Data := Value;
      end Set;
   end P;

   task body TT is
      Local : Integer := 0;
   begin
      loop
         Local := (Local + 1) mod 100;
         P.Set (Local);
      end loop;
   end TT;

end Example_04;
```

This code is correct. `Data` is part of the protected object state. The only accesses to `Data` are through `P`.

4.20.5 Example #5

Listing 22: example_05.ads

```
package Example_05 is

   protected P1 with Priority => 3 is
      procedure Set (Value : Integer);
   private
      Data : Integer := 0;
   end P1;

   protected P2 with Priority => 2 is
      procedure Set (Value : Integer);
   end P2;
```

(continues on next page)

(continued from previous page)

```
12
13 private
14    task type TT with Priority => 1;
15
16    T1, T2 : TT;
17
18 end Example_05;
```

Listing 23: example_05.adb

```
1  package body Example_05 is
2
3     protected body P1 is
4        procedure Set (Value : Integer) is
5        begin
6           Data := Value;
7        end Set;
8     end P1;
9
10    protected body P2 is
11       procedure Set (Value : Integer) is
12       begin
13          P1.Set (Value);
14       end Set;
15    end P2;
16
17    task body TT is
18       Local : constant Integer := 0;
19    begin
20       loop
21          P2.Set (Local);
22       end loop;
23    end TT;
24
25 end Example_05;
```

This code is correct. `Ceiling_Priority` policy is respected. Task never accesses a protected object with lower priority than its active priority. Note that PO can call procedure or function from another PO, but not an entry (possibly blocking).

4.20.6 Example #6

Listing 24: example_06.ads

```
1  package Example_06 is
2
3     protected type Mailbox is
4        entry Publish;
5        entry Retrieve;
6     private
7        Not_Empty    : Boolean := True;
8        Not_Full     : Boolean := False;
9        Num_Messages : Natural := 0;
10    end Mailbox;
11
12 end Example_06;
```

Listing 25: example_06.adb

```
package body Example_06 is

   Max : constant := 100;

   protected body Mailbox is
      entry Publish when Not_Full is
      begin
         Num_Messages := Num_Messages + 1;
         Not_Empty := True;
         if Num_Messages = Max then
            Not_Full := False;
         end if;
      end Publish;

      entry Retrieve when Not_Empty is
      begin
         Num_Messages := Num_Messages - 1;
         Not_Full := True;
         if Num_Messages = 0 then
            Not_Empty := False;
         end if;
      end Retrieve;
   end Mailbox;

end Example_06;
```

This code is not correct. Integer range cannot be proved correct.

4.20.7 Example #7

Listing 26: example_07.ads

```
package Example_07 is

   protected type Mailbox is
      entry Publish;
      entry Retrieve;
   private
      Num_Messages : Natural := 0;
   end Mailbox;

end Example_07;
```

Listing 27: example_07.adb

```
package body Example_07 is

   Max : constant := 100;

   protected body Mailbox is
      entry Publish when Num_Messages < Max is
      begin
         Num_Messages := Num_Messages + 1;
      end Publish;

      entry Retrieve when Num_Messages > 0 is
      begin
```

(continues on next page)

(continued from previous page)

```
13          Num_Messages := Num_Messages - 1;
14       end Retrieve;
15    end Mailbox;
16
17 end Example_07;
```

This code is correct. Precise range obtained from entry guards allows to prove checks.

4.20.8 Example #8

Listing 28: example_08.ads

```
1  package Example_08 is
2
3     Max : constant := 100;
4
5     type Content is record
6        Not_Empty    : Boolean := False;
7        Not_Full     : Boolean := True;
8        Num_Messages : Natural := 0;
9     end record with Predicate =>
10      Num_Messages in 0 .. Max
11      and Not_Empty = (Num_Messages > 0)
12      and Not_Full = (Num_Messages < Max);
13
14    protected type Mailbox is
15       entry Publish;
16       entry Retrieve;
17    private
18       C : Content;
19    end Mailbox;
20
21 end Example_08;
```

Listing 29: example_08.adb

```
1  package body Example_08 is
2
3     protected body Mailbox is
4        entry Publish when C.Not_Full is
5           Not_Full     : Boolean := C.Not_Full;
6           Num_Messages : Natural := C.Num_Messages;
7        begin
8           Num_Messages := Num_Messages + 1;
9           if Num_Messages = Max then
10             Not_Full := False;
11          end if;
12          C := (True, Not_Full, Num_Messages);
13       end Publish;
14
15       entry Retrieve when C.Not_Empty is
16          Not_Empty    : Boolean := C.Not_Empty;
17          Num_Messages : Natural := C.Num_Messages;
18       begin
19          Num_Messages := Num_Messages - 1;
20          if Num_Messages = 0 then
21             Not_Empty := False;
22          end if;
23          C := (Not_Empty, True, Num_Messages);
```

(continues on next page)

(continued from previous page)

```
      end Retrieve;
   end Mailbox;

end Example_08;
```

This code is correct. Precise range obtained from predicate allows to prove checks. Predicate is preserved.

4.20.9 Example #9

Listing 30: example_09.ads

```
--% src_file: Example_09.ads
--% cflags: -gnaty
--% make_flags: -gnaty -gnata

package Example_09 is

   package Service with
     Abstract_State => (State with External)
   is
      procedure Extract (Data : out Integer) with
        Global => (In_Out => State);
   end Service;

private
   task type T;
   T1, T2 : T;

end Example_09;
```

Listing 31: example_09.adb

```
package body Example_09 is

   package body Service with
     Refined_State => (State => Extracted)
   is
      Local_Data : constant Integer := 100;
      Extracted  : Boolean := False;

      procedure Extract (Data : out Integer) is
      begin
         if not Extracted then
            Data := Local_Data;
            Extracted := True;
         else
            Data := Integer'First;
         end if;
      end Extract;
   end Service;

   task body T is
      X : Integer;
   begin
      loop
         Service.Extract (X);
      end loop;
```

(continues on next page)

(continued from previous page)

```
26    end T;
27
28 end Example_09;
```

This code is not correct. Unsynchronized state cannot be accessed from multiple tasks or protected objects.

4.20.10 Example #10

Listing 32: example_10.ads

```
1  package Example_10 is
2
3     package Service with
4       Abstract_State => (State with Synchronous, External)
5     is
6        procedure Extract (Data : out Integer) with
7          Global => (In_Out => State);
8     private
9        protected type Service_Extracted is
10          procedure Set;
11          function Get return Boolean;
12       private
13          Extracted : Boolean := False;
14       end Service_Extracted;
15    end Service;
16
17 private
18    task type T;
19    T1, T2 : T;
20
21 end Example_10;
```

Listing 33: example_10.adb

```
1  package body Example_10 is
2
3     package body Service with
4       Refined_State => (State => Extracted)
5     is
6        Local_Data : constant Integer := 100;
7
8        Extracted : Service_Extracted;
9
10       protected body Service_Extracted is
11          procedure Set is
12          begin
13             Extracted := True;
14          end Set;
15
16        function Get return Boolean is
17           (Extracted);
18       end Service_Extracted;
19
20       procedure Extract (Data : out Integer) is
21          Is_Extracted : constant Boolean := Extracted.Get;
22       begin
23          if not Is_Extracted then
```

(continues on next page)

(continued from previous page)

```
24             Data := Local_Data;
25             Extracted.Set;
26          else
27             Data := Integer'First;
28          end if;
29       end Extract;
30    end Service;
31
32    task body T is
33       X : Integer;
34    begin
35       loop
36          Service.Extract (X);
37       end loop;
38    end T;
39
40 end Example_10;
```

This code is correct. Abstract state is synchronized, hence can be accessed from multiple tasks and protected objects.

CHAPTER

FIVE

OBJECT-ORIENTED PROGRAMMING

5.1 What is Object Oriented Programming?

> Object-oriented software construction is the building of software systems as structured collections of [...] abstract data type implementations.
>
> Bertrand Meyer, "Object Oriented Software Construction"

- Object Oriented Programming is about:
  - isolating clients from implementation details (abstraction)
  - isolating clients from the choice of data types (dynamic dispatching)
- Object Oriented Programming is not:
  - the same as prototype programming (class and objects)
  - the same as scoping (class as the scope for methods)
  - the same as code reuse (use a component in a record in SPARK)

5.2 Prototypes and Scopes in SPARK

- Types in SPARK come with methods aka primitive operations

Listing 1: show_type_primitives.ads

```
1  package Show_Type_Primitives is
2
3     type Int is range 1 .. 10;
4     function Equal (Arg1, Arg2 : Int) return Boolean;
5     procedure Bump (Arg : in out Int);
6
7     type Approx_Int is new Int;
8     --  implicit definition of Equal and Bump for Approx_Int
9
10 end Show_Type_Primitives;
```

- Scope for the prototype is current declarative region
  - ... or up to the first freezing point – point at which the type must be fully defined, e.g. when defining an object of the type
- OOP without dynamic dispatching = Abstract Data Types

5.3 Classes in SPARK

- Classes in SPARK are tagged records

Listing 2: show_classes.ads

```
package Show_Classes is

   type Int is tagged record
      Min, Max, Value : Integer;
   end record;

   function Equal (Arg1, Arg2 : Int) return Boolean;
   procedure Bump (Arg : in out Int);

   type Approx_Int is new Int with record
      Precision : Natural;
   end record;
   --  implicit definition of Equal and Bump for Approx_Int

end Show_Classes;
```

- Derived types are specializations of the root type
  - typically with more components
  - inheriting the methods on the parent type
  - can add their own methods

5.4 Methods in SPARK

- Derived methods can be overriding or not

Listing 3: show_derived_methods.ads

```
package Show_Derived_Methods is

   pragma Elaborate_Body;

   type Int is tagged record
      Min, Max, Value : Integer := 0;
   end record;

   function Equal (Arg1, Arg2 : Int) return Boolean;
   procedure Bump (Arg : in out Int);

   type Approx_Int is new Int with record
      Precision : Natural := 0;
   end record;

   overriding function Equal (Arg1, Arg2 : Approx_Int)
                              return Boolean;
   overriding procedure Bump (Arg : in out Approx_Int);

   not overriding procedure Blur (Arg : in out Approx_Int);

end Show_Derived_Methods;
```

Listing 4: show_derived_methods.adb

```
package body Show_Derived_Methods is

   function Equal (Arg1, Arg2 : Int) return Boolean is
     (Arg1 = Arg2);

   procedure Bump (Arg : in out Int) is
      Next : constant Integer := (if Arg.Value < Integer'Last
                                  then Arg.Value + 1
                                  else Integer'Last);
   begin
      if Next <= Arg.Max then
         Arg.Value := Next;
      end if;
   end Bump;

   overriding function Equal (Arg1, Arg2 : Approx_Int)
                              return Boolean is
     (Arg1 = Arg2);

   overriding procedure Bump (Arg : in out Approx_Int) is
   begin
      Bump (Int (Arg));
   end Bump;

   not overriding procedure Blur (Arg : in out Approx_Int) is
      Prev : constant Integer := (if Arg.Value > Integer'First
                                  then Arg.Value - 1
                                  else Integer'First);
   begin
      if Arg.Value >= Prev then
         Arg.Value := Prev;
      end if;
   end Blur;

end Show_Derived_Methods;
```

- Method called depends on static type

Listing 5: use_derived_methods.adb

```
with Show_Derived_Methods; use Show_Derived_Methods;

procedure Use_Derived_Methods is
   I  : Int;
   AI : Approx_Int;
begin
   Bump (I); -- call to Int.Bump
   I.Bump; -- call to Int.Bump (object.method notation)

   Bump (AI); -- call to Approx_Int.Bump
   Bump (Int (AI)); -- call to Int.Bump
end Use_Derived_Methods;
```

5.5 Dynamic dispatching in SPARK

- Class-wide types
  - type of object that triggers dispatching
  - method called depends on dynamic type

Listing 6: use_dynamic_dispatching.adb

```
with Show_Derived_Methods; use Show_Derived_Methods;

procedure Use_Dynamic_Dispatching is

   I  : Int;
   AI : Approx_Int;
begin
   declare
      IC : Int'Class := Int'Class (I);
   begin
      IC.Bump; -- call to Int.Bump
   end;

   declare
      IC : Int'Class := Int'Class (AI);
   begin
      IC.Bump; -- call to Approx_Int.Bump
   end;
end Use_Dynamic_Dispatching;
```

- Class-wide views of objects
  - in Ada, usually manipulated through pointers
  - in SPARK, manipulated through parameter passing

Listing 7: use_classwide_dispatching.adb

```
with Show_Derived_Methods; use Show_Derived_Methods;

procedure Use_Classwide_Dispatching is

   procedure Call_Bump (Arg : in out Int'Class) is
   begin
      Arg.Bump;
```

(continues on next page)

(continued from previous page)

```
   end Call_Bump;

   I  : Int;
   AI : Approx_Int;

begin
   Call_Bump (Int'Class (I));  -- calls Int.Bump(I)
   Call_Bump (Int'Class (AI)); -- calls Approx_Int.Bump(AI)
end Use_Classwide_Dispatching;
```

5.5.1 A trivial example

- what is called here?

Listing 8: show_trivial_example.adb

```
procedure Show_Trivial_Example is

   package Pkg_Trivial is
      type Int is tagged record
         Min, Max, Value : Integer;
      end record;

      procedure Bump (Arg : in out Int) is null;
   end Pkg_Trivial;

   use Pkg_Trivial;

   procedure Call_Bump
     (Arg : in out Int'Class) is
   begin
      Arg.Bump;
   end Call_Bump;

begin
   null;
end Show_Trivial_Example;
```

5.5.2 The problems with dynamic dispatching

- Control and data flow are not known statically
  - control flow – which subprogram is called when dispatching
  - data flow – what data this subprogram is accessing
  - similar to callbacks through subprogram pointers
- Avionics standard DO-178C lists 3 verification options
  - run all tests on parent type where derived type is used instead
  - cover all possible methods at dispatching calls
  - prove type substitutability (Liskov Substitution Principle aka LSP)

5.6 LSP – the SPARK solution to dynamic dispatching problems

- Class-wide contracts on methods
  - `Pre'Class` specifies strongest precondition for the hierarchy
  - `Post'Class` specifies weakest postcondition for the hierarchy

Listing 9: show_lsp.ads

```
package Show_LSP is

   type Int is tagged record
      Min, Max, Value : Integer := 0;
   end record;

   procedure Bump (Arg : in out Int) with
     Pre'Class  => Arg.Value < Arg.Max - 10,
     Post'Class => Arg.Value > Arg.Value'Old;

   type Approx_Int is new Int with record
      Precision : Natural := 0;
   end record;

   overriding procedure Bump (Arg : in out Approx_Int) with
     Pre'Class  => Arg.Value > 100,
     Post'Class => Arg.Value = Arg.Value'Old;

end Show_LSP;
```

Listing 10: show_lsp.ads

```
package Show_LSP is

   type Int is tagged record
      Min, Max, Value : Integer := 0;
   end record;

   procedure Bump (Arg : in out Int) with
     Pre'Class  => Arg.Value < Arg.Max - 10,
     Post'Class => Arg.Value > Arg.Value'Old;

   type Approx_Int is new Int with record
      Precision : Natural := 0;
   end record;

   overriding procedure Bump (Arg : in out Approx_Int) with
     Pre'Class  => True,
     Post'Class => Arg.Value = Arg.Value'Old + 10;

end Show_LSP;
```

Listing 11: show_lsp.ads

```
package Show_LSP is

   type Int is tagged record
      Min, Max, Value : Integer := 0;
   end record;

```

(continues on next page)

(continued from previous page)

```
   procedure Bump (Arg : in out Int) with
     Pre'Class  => Arg.Value < Arg.Max - 10,
     Post'Class => Arg.Value > Arg.Value'Old;

   type Approx_Int is new Int with record
      Precision : Natural := 0;
   end record;

   overriding procedure Bump (Arg : in out Approx_Int);
   --  inherited Pre'Class from Int.Bump
   --  inherited Post'Class from Int.Bump

end Show_LSP;
```

5.6.1 Verification of dynamic dispatching calls

- Class-wide contracts used for dynamic dispatching calls

Listing 12: show_dynamic_dispatching_verification.adb

```
with Show_LSP; use Show_LSP;

procedure Show_Dynamic_Dispatching_Verification is

   procedure Call_Bump (Arg : in out Int'Class) with
     Pre  => Arg.Value < Arg.Max - 10,
     Post => Arg.Value > Arg.Value'Old
   is
   begin
      Arg.Bump;
   end Call_Bump;

begin
   null;
end Show_Dynamic_Dispatching_Verification;
```

- LSP applies to data dependencies too
  - overriding method cannot read more global variables
  - overriding method cannot write more global variables
  - overriding method cannot have new input-output flows
  - SPARK RM defines `Global'Class` and `Depends'Class` (not yet implemented ⟶ use `Global` and `Depends` instead)

5.6.2 Class-wide contracts and data abstraction

- Abstraction can be used in class-wide contracts
- Typically use expression functions for abstraction

Listing 13: show_classwide_contracts.ads

```
package Show_Classwide_Contracts is

   type Int is tagged private;

```

(continues on next page)

(continued from previous page)

```
   function Get_Value (Arg : Int) return Integer;

   function Small (Arg : Int) return Boolean with Ghost;

   procedure Bump (Arg : in out Int) with
     Pre'Class  => Arg.Small,
     Post'Class => Arg.Get_Value > Arg.Get_Value'Old;

private

   type Int is tagged record
      Min, Max, Value : Integer := 0;
   end record;

   function Get_Value (Arg : Int) return Integer is
     (Arg.Value);
   function Small (Arg : Int) return Boolean is
     (Arg.Value < Arg.Max - 10);

end Show_Classwide_Contracts;
```

5.6.3 Class-wide contracts, data abstraction and overriding

- Abstraction functions can be overridden freely
  - overriding needs not be weaker or stronger than overridden

Listing 14: show_contract_override.ads

```
package Show_Contract_Override is

   type Int is tagged record
      Min, Max, Value : Integer := 0;
   end record;

   function Small (Arg : Int) return Boolean is
     (Arg.Value < Arg.Max - 10);

   type Approx_Int is new Int with record
      Precision : Natural := 0;
   end record;

   overriding function Small (Arg : Approx_Int) return Boolean is
     (True);

end Show_Contract_Override;
```

Listing 15: show_contract_override.ads

```
package Show_Contract_Override is

   type Int is tagged record
      Min, Max, Value : Integer := 0;
   end record;

   function Small (Arg : Int) return Boolean is
     (Arg.Value < Arg.Max - 10);

   type Approx_Int is new Int with record
```

(continues on next page)

(continued from previous page)

```
      Precision : Natural := 0;
   end record;

   function Small (Arg : Approx_Int) return Boolean is
     (Arg.Value in 1 .. 100);

end Show_Contract_Override;
```

- Inherited contract reinterpreted for derived class

Listing 16: show_contract_override.ads

```
package Show_Contract_Override is

   type Int is tagged record
      Min, Max, Value : Integer := 0;
   end record;

   procedure Bump (Arg : in out Int) with
     Pre'Class  => Arg.Value < Arg.Max - 10,
     Post'Class => Arg.Value > Arg.Value'Old;

   type Approx_Int is new Int with record
      Precision : Natural := 0;
   end record;

   overriding procedure Bump (Arg : in out Approx_Int);
   --  inherited Pre'Class uses Approx_Int.Small
   --  inherited Post'Class uses Approx_Int.Get_Value

end Show_Contract_Override;
```

5.7 Dynamic semantics of class-wide contracts

- Class-wide precondition is the disjunction (or) of
  - own class-wide precondition, and
  - class-wide preconditions of all overridden methods
- Class-wide postcondition is the conjunction (and) of
  - own class-wide postcondition, and
  - class-wide postconditions of all overridden methods
- Plain `Post` + class-wide `Pre` / `Post` can be used together
- Proof guarantees no violation of contracts at runtime
  - LSP guarantees stronger than dynamic semantics

5.8 Redispatching and Extensions_Visible aspect

- Redispatching is dispatching after class-wide conversion
  - formal parameter cannot be converted to class-wide type when Extensions_Visible is **False**

Listing 17: show_redispatching.adb

```
with Show_Contract_Override; use Show_Contract_Override;

procedure Show_Redispatching is

   procedure Re_Call_Bump (Arg : in out Int) is
   begin
      Int'Class (Arg).Bump;
   end Re_Call_Bump;
begin
   null;

end Show_Redispatching;
```

- Aspect Extensions_Visible allows class-wide conversion
  - parameter mode used also for hidden components

Listing 18: show_redispatching.adb

```
with Show_Contract_Override; use Show_Contract_Override;

procedure Show_Redispatching is

   procedure Re_Call_Bump (Arg : in out Int)
     with Extensions_Visible is
   begin
      Int'Class (Arg).Bump;
   end Re_Call_Bump;
begin
   null;

end Show_Redispatching;
```

5.9 Code Examples / Pitfalls

5.9.1 Example #1

Listing 19: oo_example_01.ads

```
package OO_Example_01 is

   type Int is record
      Min, Max, Value : Integer;
   end record;

   procedure Bump (Arg : in out Int) with
     Pre'Class  => Arg.Value < Arg.Max - 10,
     Post'Class => Arg.Value > Arg.Value'Old;
```

(continues on next page)

(continued from previous page)

```
10
11 end OO_Example_01;
```

This code is not correct. Class-wide contracts are only allowed on tagged records.

5.9.2 Example #2

Listing 20: oo_example_02.ads

```
1  package OO_Example_02 is
2
3     type Int is tagged record
4        Min, Max, Value : Integer;
5     end record;
6
7     procedure Bump (Arg : in out Int) with
8       Pre  => Arg.Value < Arg.Max - 10,
9       Post => Arg.Value > Arg.Value'Old;
10
11 end OO_Example_02;
```

This code is not correct. Plain precondition on dispatching subprogram is not allowed in SPARK. Otherwise it would have to be both weaker and stronger than the class-wide precondition (because they are both checked dynamically on both plain calls and dispatching calls).

Plain postcondition is allowed, and should be stronger than class-wide postcondition (plain postcondition used for plain calls).

5.9.3 Example #3

Listing 21: oo_example_03.ads

```
1  package OO_Example_03 is
2
3     pragma Elaborate_Body;
4
5     type Int is tagged record
6        Min, Max, Value : Integer;
7     end record;
8
9     procedure Bump (Arg : in out Int) with
10      Pre'Class  => Arg.Value < Arg.Max - 10,
11      Post'Class => Arg.Value > Arg.Value'Old;
12
13    type Approx_Int is new Int with record
14       Precision : Natural := 0;
15    end record;
16
17    overriding procedure Bump (Arg : in out Approx_Int) with
18      Post'Class => Arg.Value = Arg.Value'Old + 10;
19
20 end OO_Example_03;
```

Listing 22: oo_example_03.adb

```
package body OO_Example_03 is

   procedure Bump (Arg : in out Int) is
   begin
      Arg.Value := Arg.Value + 10;
   end Bump;

   overriding procedure Bump (Arg : in out Approx_Int) is
   begin
      Arg.Value := Arg.Value + 10;
   end Bump;

end OO_Example_03;
```

This code is correct. Class-wide precondition of `Int.Bump` is inherited by `Approx_Int.Bump`. Class-wide postcondition of `Approx_Int.Bump` is stronger than the one of `Int.Bump`.

5.9.4 Example #4

Listing 23: oo_example_04.ads

```
package OO_Example_04 is

   type Int is tagged record
      Min, Max, Value : Integer;
   end record;

   function "+" (Arg1, Arg2 : Int) return Int with
     Pre'Class => Arg1.Min = Arg2.Min
                  and Arg1.Max = Arg2.Max;

   type Approx_Int is new Int with record
      Precision : Natural;
   end record;

   --  inherited function “+”

end OO_Example_04;
```

This code is not correct. A type must be declared abstract or `"+"` overridden.

5.9.5 Example #5

Listing 24: oo_example_05.ads

```
package OO_Example_05 is

   type Int is tagged record
      Min, Max, Value : Integer;
   end record;

   procedure Reset (Arg : out Int);

   type Approx_Int is new Int with record
      Precision : Natural;
```

(continues on next page)

(continued from previous page)

```
   end record;

   --  inherited procedure Reset

end OO_Example_05;
```

This code is not correct. A type must be declared abstract or Reset overridden Reset is subject to Extensions_Visible **False**.

5.9.6 Example #6

Listing 25: oo_example_06.ads

```
package OO_Example_06 is

   type Int is tagged record
      Min, Max, Value : Integer;
   end record;

   procedure Reset (Arg : out Int) with Extensions_Visible;

   type Approx_Int is new Int with record
      Precision : Natural;
   end record;

   --  inherited procedure Reset

end OO_Example_06;
```

Listing 26: oo_example_06.adb

```
package body OO_Example_06 is

   procedure Reset (Arg : out Int) is
   begin
      Arg := Int'(Min   => -100,
                  Max   => 100,
                  Value => 0);
   end Reset;

end OO_Example_06;
```

This code is not correct. High: extension of Arg is not initialized in Reset.

5.9.7 Example #7

Listing 27: oo_example_07.ads

```
package OO_Example_07 is

   pragma Elaborate_Body;

   type Int is tagged record
      Min, Max, Value : Integer;
   end record;

```

(continues on next page)

(continued from previous page)

```
   function Zero return Int;

   procedure Reset (Arg : out Int) with Extensions_Visible;

   type Approx_Int is new Int with record
      Precision : Natural;
   end record;

   overriding function Zero return Approx_Int;

   --  inherited procedure Reset

end OO_Example_07;
```

Listing 28: oo_example_07.adb

```
package body OO_Example_07 is

   function Zero return Int is
      ((0, 0, 0));

   procedure Reset (Arg : out Int) is
   begin
      Int'Class (Arg) := Zero;
   end Reset;

   function Zero return Approx_Int is
       ((0, 0, 0, 0));

end OO_Example_07;
```

This code is correct. Redispatching ensures that Arg is fully initialized on return.

5.9.8 Example #8

Listing 29: file_system.ads

```
package File_System is

   type File is tagged private;

   function Closed (F : File) return Boolean;
   function Is_Open (F : File) return Boolean;

   procedure Create (F : out File) with
     Post'Class => F.Closed;

   procedure Open_Read (F : in out File) with
     Pre'Class  => F.Closed,
     Post'Class => F.Is_Open;

   procedure Close (F : in out File) with
     Pre'Class  => F.Is_Open,
     Post'Class => F.Closed;

private
   type File is tagged record
      Closed  : Boolean := True;
      Is_Open : Boolean := False;
```

(continues on next page)

(continued from previous page)

```
   end record;

   function Closed (F : File) return Boolean is
     (F.Closed);

   function Is_Open (F : File) return Boolean is
     (F.Is_Open);

end File_System;
```

Listing 30: file_system.adb

```
package body File_System is

   procedure Create (F : out File) is
   begin
      F.Closed  := True;
      F.Is_Open := False;
   end Create;

   procedure Open_Read (F : in out File) is
   begin
      F.Is_Open := True;
   end Open_Read;

   procedure Close (F : in out File) is
   begin
      F.Closed := True;
   end Close;

end File_System;
```

Listing 31: oo_example_08.adb

```
with File_System; use File_System;

procedure OO_Example_08 is

   procedure Use_File_System (F : out File'Class) is
   begin
      F.Create;
      F.Open_Read;
      F.Close;
   end Use_File_System;

begin
   null;
end OO_Example_08;
```

This code is correct. State automaton encoded in class-wide contracts is respected.

5.9.9 Example #9

Listing 32: file_system-sync.ads

```
package File_System.Sync is

   type File is new File_System.File with private;

   function Is_Synchronized (F : File) return Boolean;

   procedure Create (F : out File) with
     Post'Class => F.Closed;

   procedure Open_Read (F : in out File) with
     Pre'Class  => F.Closed,
     Post'Class => F.Is_Open and F.Is_Synchronized;

   procedure Close (F : in out File) with
     Pre'Class  => F.Is_Open and F.Is_Synchronized,
     Post'Class => F.Closed;

private
   type File is new File_System.File with record
      In_Synch : Boolean := True;
   end record;

   function Is_Synchronized (F : File) return Boolean is
     (F.In_Synch);

end File_System.Sync;
```

Listing 33: file_system-sync.adb

```
package body File_System.Sync is

   procedure Create (F : out File) is
   begin
      File_System.File (F).Create;
      F.In_Synch := True;
   end Create;

   procedure Open_Read (F : in out File) is
   begin
      File_System.File (F).Open_Read;
      F.In_Synch := True;
   end Open_Read;

   procedure Close (F : in out File) is
   begin
      File_System.File (F).Close;
      F.Closed := True;
   end Close;

end File_System.Sync;
```

Listing 34: oo_example_09.adb

```
with File_System.Sync; use File_System.Sync;

procedure OO_Example_09 is

```

(continues on next page)

(continued from previous page)

```
   procedure Use_File_System (F : out File'Class) is
   begin
      F.Create;
      F.Open_Read;
      F.Close;
   end Use_File_System;

begin
   null;
end OO_Example_09;
```

This code is not correct. Medium: class-wide precondition might be stronger than overridden one

5.9.10 Example #10

Listing 35: file_system-sync.ads

```
package File_System.Sync is

   type File is new File_System.File with private;

   function Is_Synchronized (F : File) return Boolean;

   procedure Create (F : out File) with
     Post'Class => F.Closed;

   procedure Open_Read (F : in out File) with
     Pre'Class  => F.Closed,
     Post'Class => F.Is_Open;

   procedure Close (F : in out File) with
     Pre'Class  => F.Is_Open,
     Post'Class => F.Closed;

private
   type File is new File_System.File with record
      In_Synch : Boolean;
   end record with
     Predicate => File_System.File (File).Closed
                  or In_Synch;

   function Is_Synchronized (F : File) return Boolean is
     (F.In_Synch);

end File_System.Sync;
```

Listing 36: file_system-sync.adb

```
package body File_System.Sync is

   procedure Create (F : out File) is
   begin
      File_System.File (F).Create;
      F.In_Synch := True;
   end Create;

   procedure Open_Read (F : in out File) is
```

(continues on next page)

(continued from previous page)

```
   begin
      File_System.File (F).Open_Read;
      F.In_Synch := True;
   end Open_Read;

   procedure Close (F : in out File) is
   begin
      File_System.File (F).Close;
      F.Closed := True;
   end Close;

end File_System.Sync;
```

Listing 37: oo_example_10.adb

```
with File_System.Sync; use File_System.Sync;

procedure OO_Example_10 is

   procedure Use_File_System (F : out File'Class) is
   begin
      F.Create;
      F.Open_Read;
      F.Close;
   end Use_File_System;

begin
   null;
end OO_Example_10;
```

This code is correct. Predicate encodes the additional constraint on opened files. Type invariants are not yet supported on tagged types in SPARK.

CHAPTER

SIX

GHOST CODE

6.1 What is ghost code?

ghost code is part of the program that is added for the purpose of specification

Why3 team, “The Spirit of Ghost Code”

... or verification

addition by SPARK team

- Examples of ghost code:
  - contracts (`Pre`, `Post`, `Contract_Cases`, etc.)
  - assertions (**pragma** *Assert*, loop (in)variants, etc.)
  - special values `Func'Result`, `Var'Old`, `Var'Loop_Entry`
- Is it enough?

6.2 Ghost code – A trivial example

- how to express it?

Listing 1: show_trivial_example.ads

```
1  package Show_Trivial_Example is
2
3     type Data_Array is array (1 .. 10) of Integer;
4
5     Data : Data_Array;
6     Free : Natural;
7
8     procedure Alloc;
9
10 end Show_Trivial_Example;
```

Listing 2: show_trivial_example.adb

```
1  package body Show_Trivial_Example is
2
3     procedure Alloc is
4     begin
5        --  some computations here
6        --
7        --  assert that Free “increases”
8        null;
```

(continues on next page)

(continued from previous page)

```
   end Alloc;

end Show_Trivial_Example;
```

6.3 Ghost variables – aka auxiliary variables

- Variables declared with aspect `Ghost`
  - declaration is discarded by compiler when ghost code ignored
- Ghost assignments to ghost variables
  - assignment is discarded by compiler when ghost code ignored

Listing 3: show_ghost_variable.ads

```
package Show_Ghost_Variable is

   type Data_Array is array (1 .. 10) of Integer;

   Data : Data_Array;
   Free : Natural;

   procedure Alloc;

end Show_Ghost_Variable;
```

Listing 4: show_ghost_variable.adb

```
package body Show_Ghost_Variable is

   procedure Alloc is
      Free_Init : Natural with Ghost;
   begin
      Free_Init := Free;
      --  some computations here
      pragma Assert (Free > Free_Init);
   end Alloc;

end Show_Ghost_Variable;
```

6.4 Ghost variables – non-interference rules

- Ghost variable cannot be assigned to non-ghost one
  - `Free := Free_Init;`
- Ghost variable cannot indirectly influence assignment to non-ghost one

```
if Free_Init < Max then
   Free := Free + 1;
end if;
```

Listing 5: show_non_interference.adb

```
procedure Show_Non_Interference is

   type Data_Array is array (1 .. 10) of Integer;

   Data : Data_Array;
   Free : Natural;

   Free_Init : Natural with Ghost;

   procedure Alloc is
   begin
      Free_Init := Free;
      --  some computations here
      pragma Assert (Free > Free_Init);
   end Alloc;

   procedure Assign (From : Natural; To : out Natural) is
   begin
      To := From;
   end Assign;

begin
   Assign (From => Free_Init, To => Free);
end Show_Non_Interference;
```

6.5 Ghost statements

- Ghost variables can only appear in ghost statements
  - assignments to ghost variables
  - assertions and contracts
  - calls to ghost procedures

Listing 6: show_ghost_statements.adb

```
procedure Show_Ghost_Statements is

   type Data_Array is array (1 .. 10) of Integer;

   Data : Data_Array;
   Free : Natural;

   Free_Init : Natural with Ghost;

   procedure Alloc is
   begin
      Free_Init := Free;
      --  some computations here
      pragma Assert (Free > Free_Init);
   end Alloc;

   procedure Assign (From : Natural; To : out Natural)
     with Ghost
   is
   begin
```

(continues on next page)

(continued from previous page)

```
      To := From;
   end Assign;

begin
   Assign (From => Free, To => Free_Init);
end Show_Ghost_Statements;
```

```
procedure Show_Ghost_Statements is
begin
   --  Non-ghost variable "Free" cannot appear as actual in
   --  call to ghost procedure
   Assign (From => Free_Init, To => Free);
end Show_Ghost_Statements;
```

6.6 Ghost procedures

- Ghost procedures cannot write into non-ghost variables

```
procedure Assign (Value : Natural) with Ghost is
begin
   --  "Free" is a non-ghost variable
   Free := Value;
end Assign;
```

- Used to group statements on ghost variables
  - in particular statements not allowed in non-ghost procedures

```
procedure Assign_Cond (Value : Natural) with Ghost is
begin
   if Condition then
      Free_Init := Value;
   end if;
end Assign_Cond;
```

- Can have `Global` (including `Proof_In`) & `Depends` contracts

6.7 Ghost functions

- Functions for queries used only in contracts
- Typically implemented as expression functions
  - in private part – proof of client code can use expression
  - or in body – only proof of unit can use expression

Listing 7: show_ghost_function.ads

```
package Show_Ghost_Function is

   type Data_Array is array (1 .. 10) of Integer;

   Data : Data_Array;
   Free : Natural;

```

(continues on next page)

(continued from previous page)

```
8     Free_Init : Natural with Ghost;
9
10    procedure Alloc with
11      Pre  => Free_Memory > 0,
12      Post => Free_Memory < Free_Memory'Old;
13
14    function Free_Memory return Natural with Ghost;
15
16 private
17
18    --  Completion of ghost function declaration
19    function Free_Memory return Natural is
20      (0); -- dummy implementation
21
22    --  If function body as declaration:
23    --
24    --     function Free_Memory return Natural is (...) with Ghost;
25
26 end Show_Ghost_Function;
```

6.8 Imported ghost functions

- Ghost functions without a body
  - cannot be executed

```
function Free_Memory return Natural with Ghost, Import;
```

- Typically used with abstract ghost private types
  - definition in SPARK_Mode(Off)
    - type is abstract for GNATprove

Listing 8: show_imported_ghost_function.ads

```
1  package Show_Imported_Ghost_Function
2    with SPARK_Mode => On is
3
4     type Memory_Chunks is private;
5
6     function Free_Memory return Natural with Ghost;
7
8     function Free_Memory return Memory_Chunks
9        with Ghost, Import;
10
11 private
12    pragma SPARK_Mode (Off);
13
14    type Memory_Chunks is null record;
15
16 end Show_Imported_Ghost_Function;
```

- Definition of ghost types/functions given in proof
  - either in Why3 using External_Axiomatization
  - or in an interactive prover (Coq, Isabelle, etc.)

6.9 Ghost packages and ghost abstract state

- Every entity in a ghost package is ghost
  - local ghost package can group all ghost entities
  - library-level ghost package can be withed/used in regular units
- Ghost abstract state can only represent ghost variables

Listing 9: show_ghost_package.ads

```
package Show_Ghost_Package
  with Abstract_State => (State with Ghost) is

   function Free_Memory return Natural with Ghost;

end Show_Ghost_Package;
```

Listing 10: show_ghost_package.adb

```
package body Show_Ghost_Package
  with Refined_State => (State => (Data, Free, Free_Init)) is

   type Data_Array is array (1 .. 10) of Integer;

   Data : Data_Array with Ghost;
   Free : Natural with Ghost;

   Free_Init : Natural with Ghost;

   function Free_Memory return Natural is
     (0);  --  dummy implementation

end Show_Ghost_Package;
```

- Non-ghost abstract state can contain both ghost and non-ghost variables

6.10 Executing ghost code

- Ghost code can be enabled globally
  - using compilation switch `-gnata` (for all assertions)
- Ghost code can be enabled selectively
  - using `pragma Assertion_Policy (Ghost => Check)`
  - SPARK rules enforce consistency – in particular no write disabled

Listing 11: show_exec_ghost_code.ads

```
package Show_Exec_Ghost_Code is

   pragma Assertion_Policy (Ghost => Check);
   --  pragma Assertion_Policy (Ghost => Ignore, Pre => Check);

   procedure Alloc with
     Pre => Free_Memory > 0;

   function Free_Memory return Natural with Ghost;
```

(continues on next page)

(continued from previous page)

```
10
11 end Show_Exec_Ghost_Code;
```

- GNATprove analyzes all ghost code and assertions

6.11 Examples of use

6.11.1 Encoding a state automaton

- Tetris in SPARK
  - at Tetris[3]
- Global state encoded in global ghost variable
  - updated at the end of procedures of the API

```
type State is (Piece_Falling, ...) with Ghost;
Cur_State : State with Ghost;
```

- Properties encoded in ghost functions

```
function Valid_Configuration return Boolean is
   (case Cur_State is
      when Piece_Falling => ...,
      when ...)
with Ghost;
```

6.11.2 Expressing useful lemmas

- GCD in SPARK
  - at GCD[4]
- Lemmas expressed as ghost procedures

```
procedure Lemma_Not_Divisor (Arg1, Arg2 : Positive) with
   Ghost,
   Global => null,
   Pre  => Arg1 in Arg2 / 2 + 1 .. Arg2 - 1,
   Post => not Divides (Arg1, Arg2);
```

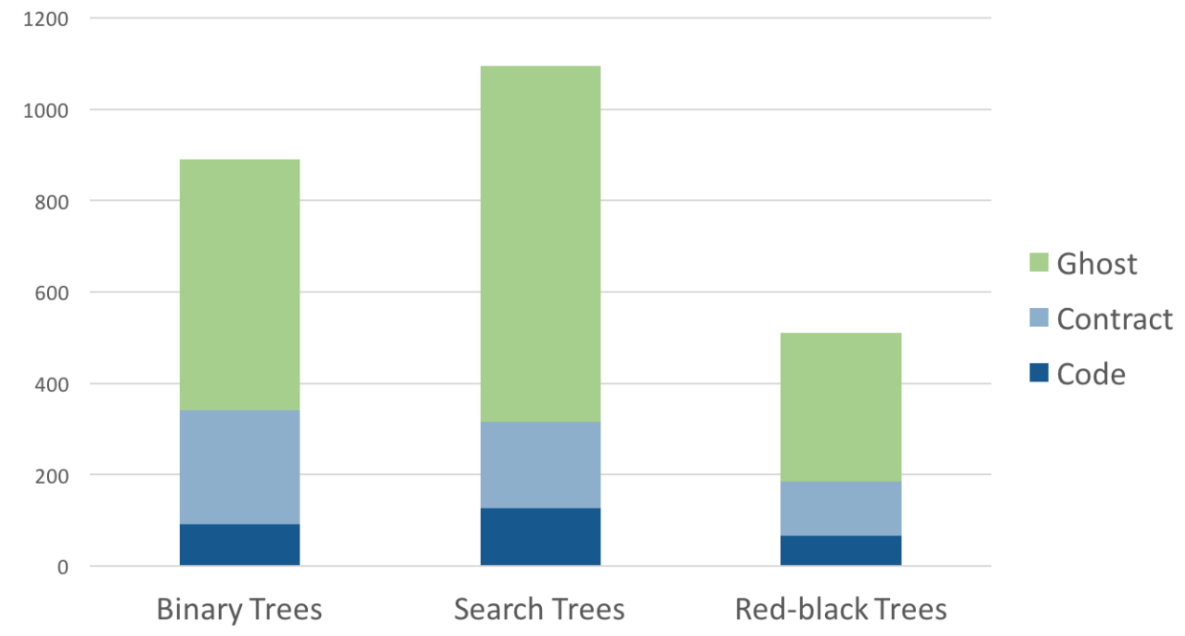
- Most complex lemmas further refined into other lemmas
  - code in procedure body used to guide proof (e.g. for induction)

[3] http://blog.adacore.com/tetris-in-spark-on-arm-cortex-m4
[4] http://www.spark-2014.org/entries/detail/gnatprove-tips-and-tricks-proving-the-ghost-common-denominator-gcd

6.11.3 Specifying an API through a model

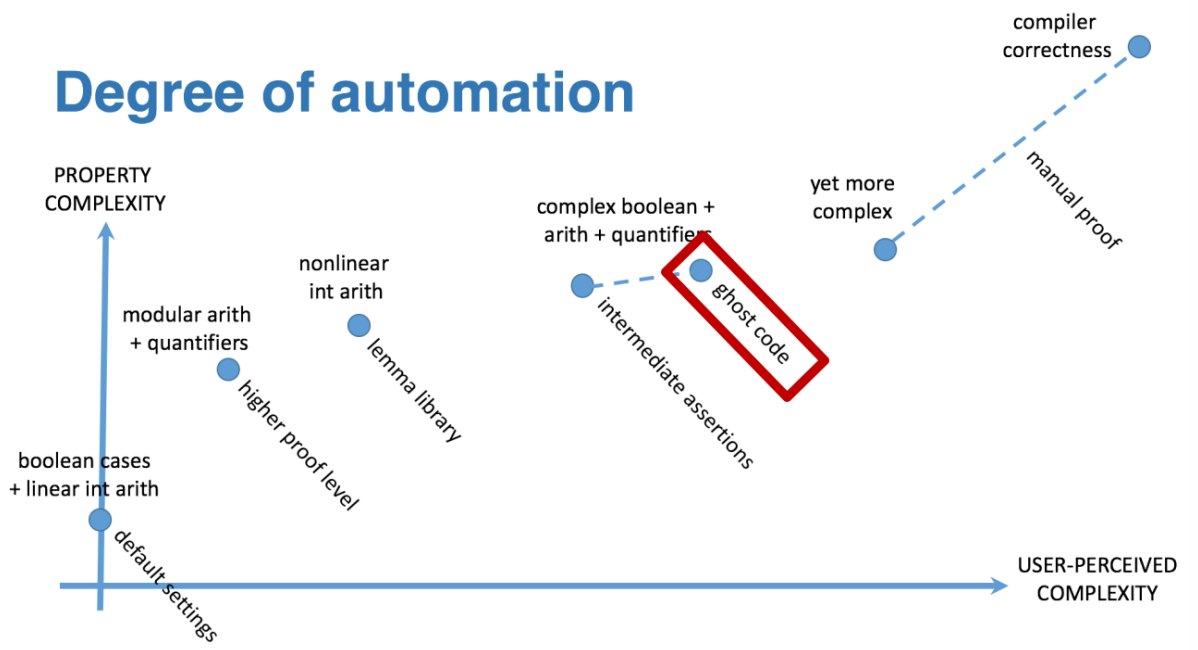
- Red black trees in SPARK
  - at Red black trees[5]
- Invariants of data structures expressed as ghost functions
  - using `Type_Invariant` on private types
- Model of data structures expressed as ghost functions
  - called from `Pre` / `Post` of subprograms from the API
- Lemmas expressed as ghost procedures
  - sometimes without contracts to benefit from inlining in proof

6.12 Extreme proving with ghost code – red black trees in SPARK



[5] http://www.spark-2014.org/entries/detail/research-corner-auto-active-verification-in-spark

6.13 Positioning ghost code in proof techniques



6.14 Code Examples / Pitfalls

6.14.1 Example #1

Listing 12: example_01.adb

```
procedure Example_01 is

   type Data_Array is array (1 .. 10) of Integer;

   Data : Data_Array;
   Free : Natural;

   procedure Alloc is
      Free_Init : Natural with Ghost;
   begin
      Free_Init := Free;
      --  some computations here
      if Free <= Free_Init then
         raise Program_Error;
      end if;
   end Alloc;
begin
   null;

end Example_01;
```

This code is not correct. A ghost entity cannot appear in this context.

6.14.2 Example #2

Listing 13: example_02.adb

```
procedure Example_02 is

   type Data_Array is array (1 .. 10) of Integer;

   Data : Data_Array;
   Free : Natural;

   procedure Alloc is
      Free_Init : Natural with Ghost;

      procedure Check with Ghost is
      begin
         if Free <= Free_Init then
            raise Program_Error;
         end if;
      end Check;
   begin
      Free_Init := Free;
      --  some computations here
      Check;
   end Alloc;
begin
   null;

end Example_02;
```

This code is correct. Note that procedure Check is inlined for proof (no contract).

6.14.3 Example #3

Listing 14: example_03.ads

```
package Example_03 is

   type Data_Array is array (1 .. 10) of Integer;

   Data : Data_Array;
   Free : Natural;

   pragma Assertion_Policy (Pre => Check);

   procedure Alloc with
     Pre => Free_Memory > 0;

   function Free_Memory return Natural with Ghost;

end Example_03;
```

This code is not correct. Incompatible ghost policies in effect during compilation, as ghost code is ignored by default. Note that GNATprove accepts this code as it enables all ghost code and assertions.

6.14.4 Example #4

Listing 15: example_04.ads

```
package Example_04 is

   procedure Alloc with
     Post => Free_Memory < Free_Memory'Old;

   function Free_Memory return Natural with Ghost;

end Example_04;
```

Listing 16: example_04.adb

```
package body Example_04 is

   Free : Natural;

   Max : constant := 1000;

   function Free_Memory return Natural is
   begin
      return Max - Free + 1;
   end Free_Memory;

   procedure Alloc is
   begin
      Free := Free + 10;
   end Alloc;

end Example_04;
```

This code is not correct. No postcondition on `Free_Memory` that would allow proving the postcondition on `Alloc`.

6.14.5 Example #5

Listing 17: example_05.ads

```
package Example_05 is

   procedure Alloc with
     Post => Free_Memory < Free_Memory'Old;

   function Free_Memory return Natural with Ghost;

end Example_05;
```

Listing 18: example_05.adb

```
package body Example_05 is

   Free : Natural;

   Max : constant := 1000;

   function Free_Memory return Natural is (Max - Free + 1);

   procedure Alloc is
   begin
      Free := Free + 10;
   end Alloc;

end Example_05;
```

This code is correct. Free_Memory has an implicit postcondition as an expression function.

6.14.6 Example #6

Listing 19: example_06.adb

```
procedure Example_06 is

   subtype Resource is Natural range 0 .. 1000;
   subtype Num is Natural range 0 .. 6;
   subtype Index is Num range 1 .. 6;
   type Data is array (Index) of Resource;

   function Sum (D : Data; To : Num) return Natural is
     (if To = 0 then 0 else D (To) + Sum (D, To - 1))
       with Ghost;

   procedure Create (D : out Data) with
     Post => Sum (D, D'Last) < 42
   is
   begin
      for J in D'Range loop
         D (J) := J;
         pragma Loop_Invariant (2 * Sum (D, J) <= J * (J + 1));
      end loop;
   end Create;

begin
   null;
end Example_06;
```

This code is not correct. Info: expression function body not available for proof (Sum may not return).

6.14.7 Example #7

Listing 20: example_07.adb

```
procedure Example_07 is

   subtype Resource is Natural range 0 .. 1000;
   subtype Num is Natural range 0 .. 6;
   subtype Index is Num range 1 .. 6;
   type Data is array (Index) of Resource;

   function Sum (D : Data; To : Num) return Natural is
     (if To = 0 then 0 else D (To) + Sum (D, To - 1))
       with Ghost, Annotate => (GNATprove, Terminating);

   procedure Create (D : out Data) with
     Post => Sum (D, D'Last) < 42
   is
   begin
      for J in D'Range loop
         D (J) := J;
         pragma Loop_Invariant (2 * Sum (D, J) <= J * (J + 1));
      end loop;
   end Create;

begin
   null;
end Example_07;
```

This code is correct. Note that GNATprove does not prove the termination of Sum here.

6.14.8 Example #8

Listing 21: example_08.adb

```
procedure Example_08 is

   subtype Resource is Natural range 0 .. 1000;
   subtype Num is Natural range 0 .. 6;
   subtype Index is Num range 1 .. 6;
   type Data is array (Index) of Resource;

   function Sum (D : Data; To : Num) return Natural is
     (if To = 0 then 0 else D (To) + Sum (D, To - 1))
       with Ghost, Annotate => (GNATprove, Terminating);

   procedure Create (D : out Data) with
     Post => Sum (D, D'Last) < 42
   is
   begin
      for J in D'Range loop
         D (J) := J;
      end loop;
   end Create;

```

(continues on next page)

(continued from previous page)

```
begin
   null;
end Example_08;
```

This code is correct. The loop is unrolled by GNATprove here, as `D'Range` is `0 .. 6`. The automatic prover unrolls the recursive definition of `Sum`.

6.14.9 Example #9

Listing 22: example_09.adb

```
with Ada.Containers.Functional_Vectors;

procedure Example_09 is

   subtype Resource is Natural range 0 .. 1000;
   subtype Index is Natural range 1 .. 42;

   package Seqs is new
     Ada.Containers.Functional_Vectors (Index, Resource);
   use Seqs;

   function Create return Sequence with
     Post => (for all K in 1 .. Last (Create'Result) =>
                 Get (Create'Result, K) = K)
   is
      S : Sequence;
   begin
      for K in 1 .. 42 loop
         S := Add (S, K);
      end loop;
      return S;
   end Create;

begin
   null;
end Example_09;
```

This code is not correct. Loop requires a loop invariant to prove the postcondition.

6.14.10 Example #10

Listing 23: example_10.adb

```
with Ada.Containers.Functional_Vectors;

procedure Example_10 is

   subtype Resource is Natural range 0 .. 1000;
   subtype Index is Natural range 1 .. 42;

   package Seqs is new
     Ada.Containers.Functional_Vectors (Index, Resource);
   use Seqs;

   function Create return Sequence with
     Post => (for all K in 1 .. Last (Create'Result) =>
```

(continues on next page)

(continued from previous page)

```
                  Get (Create'Result, K) = K)
   is
      S : Sequence;
   begin
      for K in 1 .. 42 loop
         S := Add (S, K);
         pragma Loop_Invariant (Integer (Length (S)) = K);
         pragma Loop_Invariant
           (for all J in 1 .. K => Get (S, J) = J);
      end loop;
      return S;
   end Create;

begin
   null;
end Example_10;
```

This code is correct.

CHAPTER

SEVEN

TEST AND PROOF

7.1 Various Combinations of Tests and Proofs

- Overall context is functional verification of code
- Combination can take various forms:
  - Test before Proof – contracts used first in test, possibly later in proof
  - Test for Proof – contracts executed in test to help with development of proof
  - Test alongside Proof – some modules are tested and other modules are proved
  - Test as Proof – exhaustive test as good as proof
  - Test on top of Proof – proof at unit level completed with test at integration level, also using contracts

7.2 Test (be)for(e) Proof

7.2.1 Activating Run-time Checks

- Need to activate run-time checks in executable
- `Constraint_Error` exceptions activated by default
  - Use `-gnat-p` to revert effect of previous `-gnatp` (say in project file)
  - Use `-gnato` to activate overflow checking (default since GNAT 7.3)
- Special handling of floating-point computations
  - Use `-gnateF` to activate bound checking on standard float types
  - Use `-msse2 -mfpmath=sse` to forbid use of 80bits registers and FMA on x86 processors
  - Runtime/BSP should enforce use of Round-Nearest-tie-to-Even (RNE) rounding mode

7.2.2 Activating Assertions

- Need to activate assertions in executable
- `Assertion_Error` exceptions deactivated by default
  - Use `-gnata` to activate globally
  - Use pragma `Assertion_Policy` to activate file-by-file
  - Use `-gnateE` to get more precise error messages (`Contract_Cases`)
- Special assertions checked at run time
  - `Contract_Cases` ⟶ checks one and only one case activated
  - `Loop_Invariant` ⟶ checks assertion holds (even if not inductive)
  - `Assume` ⟶ checks assertion holds (even if not subject to proof)
  - `Loop_Variant` ⟶ checks variant decreases wrt previous iteration

7.2.3 Activating Ghost Code

- Need to activate ghost code in executable
- Ghost code, like assertions, is deactivated by default
  - Use `-gnata` to activate globally
  - Use pragma `Assertion_Policy (Ghost => Check)` to activate locally
- Inconsistent combinations will be rejected by GNAT
  - Ignored ghost entity in activated assertion
  - Ignored ghost assignment to activated ghost variable

7.3 Test for Proof

7.3.1 Overflow Checking Mode

- Problem: ignore overflow checks in assertions/contracts
  - Only applies to signed integer arithmetic
  - Does not apply inside an expression function returning an integer
- Solution: use unbounded arithmetic in assertions/contracts
  - Will use 64bits signed arithmetic when sufficient
  - Otherwise use a run-time library for unbounded arithmetic
- Two ways to activate unbounded arithmetic
  - Use `-gnato13` compiler switch
  - Use pragma `Overflow_Mode` with arguments `(General => Strict, Assertions => Eliminated)` in configuration pragma file

7.4 Test alongside Proof

7.4.1 Checking Proof Assumptions

- Need to check dynamically the assumptions done in proof
  - Postcondition of tested subprogram called in proved subprogram
  - Precondition of proved subprogram called in tested subprogram
- Other assumptions beyond pre- and postconditions
  - Global variables read and written by tested subprogram
  - Non-aliasing of inputs and outputs of proved subprogram
  - No run-time errors in tested subprogram
- GNATprove can list assumptions used in proof
  - Switch `--assumptions` adds info in `gnatprove.out` file
- See "Explicit Assumptions - A Prenup for Marrying Static and Dynamic Program Verification"

7.4.2 Rules for Defining the Boundary

- `SPARK_Mode` defines a simple boundary test vs. proof
  - Subprograms with `SPARK_Mode (On)` should be proved
  - Subprograms with `SPARK_Mode (Off)` should be tested
- `SPARK_Mode` can be used at different levels
  - Project-wise switch in configuration pragma file (with value `On`) ⟶ explicit exemptions of units/subprograms in the code
  - Distinct GNAT project with `SPARK_Mode (On)` for proof on subset of units
  - Explicit `SPARK_Mode (On)` on units that should be proved
- Unproved checks inside proved subprograms are justified
  - Use of pragma `Annotate` inside the code

7.4.3 Special Compilation Switches

- Validity checking for reads of uninitialized data
  - Compilation switch `-gnatVa` enables validity checking
  - pragma `Initialize_Scalars` uses invalid default values
  - Compilation switch `-gnateV` enables validity checking for composite types (records, arrays) ⟶ extra checks to detect violation of SPARK stronger data initialization policy
- Non-aliasing checks for parameters
  - Compilation switch `-gnateA` enables non-aliasing checks between parameters
  - Does not apply to aliasing between parameters and globals

7.5 Test as Proof

7.5.1 Feasibility of Exhaustive Testing

- Exhaustive testing covers all possible input values
  - Typically possible for numerical computations involving few values
  - e.g. OK for 32 bits values, not for 64 bits ones
    - binary op on 16 bits ⟶ 1 second with 4GHz
    - unary op on 32 bits ⟶ 1 second with 4GHz
    - binary op on 32 bits ⟶ 2 years with 64 cores at 4GHz
  - In practice, this can be feasible for trigonometric functions on 32 bits floats
- Representative/boundary values may be enough
  - Partitioning of the input state in equivalent classes
  - Relies on continuous/linear behavior inside a partition

7.6 Test on top of Proof

7.6.1 Combining Unit Proof and Integration Test

- Unit Proof of AoRTE combined with Integration Test
  - Combination used by Altran UK on several projects
  - Unit Proof assumes subprogram contracts
  - Integration Test verifies subprogram contracts
- Unit Proof of Contracts combined with Integration Test
  - Test exercises the assumptions made in proof
  - One way to show Property Preservation between Source Code and Executable Object Code from DO-178C/DO-333
    - Integration Test performed twice: once with contracts to show they are verified in EOC, once without to show final executable behaves the same

7.7 Test Examples / Pitfalls

7.7.1 Example #1

I am stuck with an unproved assertion. My options are:

- switch --level to 4 and --timeout to 360
- open a ticket on GNAT Tracker
- justify the unproved check manually

Evaluation: This approach is not correct. Why not, but only after checking this last option:

- run tests to see if the assertion actually holds

7.7.2 Example #2

The same contracts are useful for test and for proof, so it's useful to develop them for test initially.

Evaluation: This approach is not correct. In fact, proof requires more contracts that test, as each subprogram is analyzed separately. But these are a superset of the contracts used for test.

7.7.3 Example #3

Assertions need to be activated explicitly at compilation for getting the corresponding run-time checks.

Evaluation: This approach is correct. Use switch `-gnata` to activate assertions.

7.7.4 Example #4

When assertions are activated, loop invariants are checked to be inductive on specific executions.

Evaluation: This approach is not correct. Loop invariants are checked dynamically exactly like assertions. The inductive property is not something that can be tested.

7.7.5 Example #5

Procedure P which is proved calls function T which is tested. To make sure the assumptions used in the proof of P are verified, we should check dynamically the precondition of T.

Evaluation: This approach is not correct. The precondition is proved at the call site of T in P. But we should check dynamically the postcondition of T.

7.7.6 Example #6

Function T which is tested calls procedure P which is proved. To make sure the assumptions used in the proof of P are verified, we should check dynamically the precondition of P.

Evaluation: This approach is correct. The proof of P depends on its precondition being satisfied at every call.

7.7.7 Example #7

However procedure P (proved) and function T (tested) call each other, we can verify the assumptions of proof by checking dynamically all preconditions and postconditions during tests of T.

Evaluation: This approach is not correct. That covers only functional contracts. There are other assumptions made in proof, related to initialization, effects and non-aliasing.

7.7.8 Example #8

Proof is superior to test in every aspect.

Evaluation: This approach is not correct. Maybe for the aspects `Pre` and `Post`. But not in other aspects of verification: non-functional verification (memory footprint, execution time), match with hardware, integration in environment... And testing can even be exhaustive sometimes!

7.7.9 Example #9

When mixing test and proof at different levels, proof should be done at unit level and test at integration level.

Evaluation: This approach is not correct. This is only one possibility that has been used in practice. The opposite could be envisioned: test low-level functionalities (e.g. crypto in hardware), and prove correct integration of low-level functionalities.

7.7.10 Example #10

There are many ways to mix test and proof, and yours may not be in these slides.

Evaluation: This approach is correct. YES! (and show me yours)