



# ALFA: Annotation Language For Ada

Yannick Moy

Hi-Lite WP2 (Specifications)  
July 13, 2010

# Objectives of this First Meeting

*clarify the boundary between what is or not in Hi-Lite's scope*

*describe the facilities to express properties in ALFA and E-ACSL*

*agree on priorities for supporting these facilities in tools*

Subprogram Contracts

Data and Control Invariants

Summary

Subprogram Contracts

Data and Control Invariants

Summary

*a coffee-machine should pour coffee only when the amount of money is sufficient and the user did not abort the process*

```
function Deliver_Dose return Boolean
  with Pre  => Price >= 0,
        Post => Deliver_Dose ' Result =
                Cash >= Price
        and not Changed_My_Mind ' Old;
```

- ▶ precondition can:
  - ▶ protect against invalid use of the function
  - ▶ enforce a protocol (a state automaton) on subprograms
- ▶ postcondition can:
  - ▶ describe properties of the post-state and result ('Result)
  - ▶ relate the post-state and the pre-state ('Old)

## Pre and Postconditions **are not** about ...

- ▶ absence of exceptions
- ▶ exceptional postcondition (like in JML)
- ▶ termination
- ▶ memory/time footprint
- ▶ read and write effects (like the global in/out of SPARK):

*PushElement is the only way to insert elements in the list*

- ▶ sequences of events:

*each time I call ProcessAllElements, all elements that were inserted using PushElement are processed*

# Test Cases as GNAT-Specific Aspect

```
function Deliver_Dose return Boolean
  with Test_Case_1 =>
    (Requires => Cash >= 0
      and Cash < Price
      and not Changed_My_Mind ,
     Ensures => not Deliver_Dose ' Result
      and not Error );
```

- ▶ same as *behavior* in JML, ACSL
  - ▶ Requires: describes pre-state like Pre
  - ▶ Ensures: relates pre-state and post-state like Post
- ▶ possible splitting of Requires part into:
  - ▶ Assumes: when the test-case applies
  - ▶ Requires: additional preconditions for this test-case
  - ▶ *if the software is in the state S, we can only receive this kind of input, and then the software computes this result*
  - ▶ precondition is split between Pre and Requires

# Specification Coverage as GNAT-Specific Aspect

```
function Deliver_Dose return Boolean  
  with Partition  $\Rightarrow$  (Cash < Price , Changed_My_Mind );
```

- ▶ partitions the pre-state along some predicates:
  - ▶ case 1: (Cash < Price) and Changed\_My\_Mind
  - ▶ case 2: (Cash < Price) and not Changed\_My\_Mind
  - ▶ case 3: not (Cash < Price) and Changed\_My\_Mind
  - ▶ case 4: not (Cash < Price) and not Changed\_My\_Mind
- ▶ equivalent to a set of test-cases with Ensures = True
- ▶ opens the possibility of combined testing/verification

```
type Ordering is (Less_Than , Equal , Greater_Than );  
function Compare (X, Y : Amount) return Ordering ;  
function Deliver_Dose return Boolean  
  with Partition  $\Rightarrow$  (Compare (Cash , Price) ,  
                    Changed_My_Mind );
```

```
function Deliver_Dose return Boolean
  with Pre => (if Changed_My_Mind then Price = 0),
       Post => (Deliver_Dose ' Result =
                case Compare (Cash, Price) is
                  when Less_Than => False
                  when Equal |
                    Greater_Than => True);
```

- ▶ conditional expressions:
  - ▶ (if A then B) instead of (not A or else B)
  - ▶ (if A then B else C) instead of (A and then B or else not A and then C) when B, C Boolean
  - ▶ B, C of same type (not only Boolean)
- ▶ case expressions:
  - ▶ all alternatives of same type (not only Boolean)
  - ▶ compiler checks exhaustivity

```
function Deliver_Dose return Boolean is  
  (Price >= 0 and Cash >= Price and not Changed_My_Mind);
```

- ▶ rules:
  - ▶ function *body* is an expression
  - ▶ no calls to regular functions, only other expression functions
  - ▶ can be defined in package specification
- ▶ benefits:
  - ▶ guaranteed without (write) side-effects
  - ▶ gives the exact read effects
  - ▶ allows exposing the precise specification
  - ▶ no need for (read/write) effect analysis
  - ▶ no need for *pure* pragma
  - ▶ no need for global annotations

# Executable Quantification in Ada 2012

*a list can be used to store the set of currently enabled monitoring of a system. Then, we may wish to verify that if no monitoring detects failure, then no recovery is triggered*

```
package Enabled_Monitorings is  
  new Ada.Containers.Doubly_Linked_Lists (Monitoring);  
  
L : Enabled_Monitorings.List;  
  
pragma Assert  
  (if (for all M in L | not Detects_Failure (M))  
    then Recovery_Mode = None);
```

- ▶ quantification over an array range
- ▶ quantification over a container
- ▶ universal (for all) and existential (for some) quantification

# Generalized Quantification

*we may wish to verify that if some monitoring detects failure, then the recovery with the highest priority is triggered*

```
package Recoveries is  
  new Ada.Containers.Ordered_Set (Recovery);  
  
R : Recoveries.Set;  
  
pragma Assert  
  (if (for some M in L | Detects_Failure (M)) then  
    Max (R, Order_By_Priority 'Access) = The_Recovery);
```

- ▶ JML  $\sum$ ,  $\prod$ ,  $\max$ ,  $\min$ :
- ▶  $\min$ ,  $\max$  expressible with existential quantification
- ▶ best handled with dedicated support
- ▶  $\sum$ ,  $\prod$  **not** expressible otherwise
- ▶ special kinds of folding operations

## Generalized Quantification (Cont'd)

*we may need to verify that in a list at most  $n$  elements are in a given state*

```
function Is_Active (M : Monitoring) return Boolean;  
pragma Assert (Num_Of (L, Is_Active 'Access) <= N);
```

- ▶ JML \num\_of
- ▶ just another special case of folding operations
- ▶ translation to logic requires a known function address as parameter

Subprogram Contracts

Data and Control Invariants

Summary

# Type Invariants

*a well-formed stack is bounded by values Top and Bottom, with Top never less than Bottom - 1*

```
package Stacks is

  type Stack is private
    with Invariant  $\Rightarrow$  Well_Formed_Stack (Stack);

private

  type S is record
    Bottom, Top : Natural;
    ...
  end record;

  function Well_Formed_Stack (S : Stack) return Boolean is
    (S.Bottom - 1  $\leq$  S.Top);

end Stack;
```

## Type Invariants (Cont'd)

- ▶ Invariants are checked:
  - ▶ on object initialization
  - ▶ on conversion to the type
  - ▶ on return from function that creates object of the type
  - ▶ on return from subprogram that has (in)- out parameter of the type
- ▶ type-specific Invariant and classwide Invariant'Class (follows Liskov)
- ▶ not bullet-proof (access values, visible components)
- ▶ hopefully match upcoming type invariants in SPARK!

## Subtype Predicates

*an even number is a natural number divisible by 2*

*a t-day is a day whose English name begins by 't'*

*a weekend-schedule is a schedule for a weekend day*

```
subtype Even is Natural
  with Predicate  $\Rightarrow$  (Even mod 2 = 0);

type Day is (Monday, Tuesday, Wednesday, Thursday,
              Friday, Saturday, Sunday);
subtype T_Day is Day
  with Predicate  $\Rightarrow$  (T_Day in Tuesday | Thursday);

type Schedule (D : Day) is record ... end record;
subtype Weekend_Schedule is Schedule
  with Weekend_Schedule.D in Saturday .. Sunday;
```

## Subtype Predicates (Cont'd)

- ▶ Predicates are checked:
  - ▶ on every subtype conversion (assignments)
  - ▶ on all parameter passing
- ▶ applicable to any subtype
- ▶ most common use: non-contiguous enumeration types
- ▶ not same as Ada constraints (range, not null): *constraints can only be violated for invalid values, whereas predicates can be violated in various ways*

## N<sup>o</sup> 1 problem in proving programs!

- ▶ loop invariant must be:
  - ▶ *true*: like an assertion
  - ▶ *inductive*: provable from both previous context of loop (first iteration) or from itself at previous iteration
  - ▶ *strong enough*: sufficient to prove code that follows
  - ▶ *simple enough*: to be written by the programmer
- ▶ loop invariant subtleties:
  - ▶ depends on VC generation: some tools (Why) do not require to state properties of variable not modified in the loop, some do (Examiner)
  - ▶ checked either at loop head or inside loop body (SPARK allows both)

```
pragma Loop_Invariant (...); — ???
```

# Loop Invariant for Containers

*a loop iterates over a set to increment its values*

```
declare
  Cont : Set;
  Cur  : Cursor := First (Cont);
begin
  while Has_Element (Cur) loop
    pragma Assert (for all X in Cont range
                  First (Cont) .. Previous (Cur) |
                  X = X'Old + 1);
    pragma Assert (for all X in Right (Cont, Cur) |
                  X = X'Old);
    Replace_Element (Cont, Cur, Element (Cur) + 1);
    Next (Cur);
  end loop;
end;
```

# New API for Containers

- ▶ remove unprovable functionalities:
  - ▶ subprograms taking access procedures as parameters (Query\_Element, Update\_Element, Iterate)
  - ▶ prevents any “tampering”
  - ▶ replaced by direct use of cursors
- ▶ new model for cursors:
  - ▶ a simple “index” in the physical array of data
  - ▶ need to pass the container as parameter for reads too
- ▶ need richer API for loop invariants:
  - ▶ position of cursors
  - ▶ validity of cursors
  - ▶ “left” and “right” sub-containers
  - ▶ new functions are **terribly inefficient!** typically only for annotations, not code

Subprogram Contracts

Data and Control Invariants

Summary

- ▶ Syntax for aspects on declarations
- ▶ Pre and postconditions on subprograms
- ▶ Conditional expressions
- ▶ Case expressions
- ▶ Parameterized expressions
- ▶ Quantified expressions (over array or container)
- ▶ Invariants on private types
- ▶ Predicates on subtypes

- ▶ Test cases on subprograms
- ▶ Partitioning of inputs on subprograms
- ▶ Min, Max, Sum and Num\_Of in API of containers
- ▶ Loop invariant as pragma
- ▶ New API for containers (for proofs)