John Barnes

Ada Rationale 2012

Epilogue

Courtesy of

AdaCore

Rationale for Ada 2012: Epilogue

John Barnes
John Barnes Informatics, 11 Albert Road, Caversham, Reading RG4 7AN, UK; Tel: +44 118 947 4125; email: jgpb@jbinfo.demon.co.uk

Abstract

This is the last of a number of papers describing the rationale for Ada 2012. In due course it is anticipated that the papers will be combined (after appropriate reformatting and editing) into a single volume for formal publication.

This last paper summarizes a small number of general issues of importance to the user such as compatibility between Ada 2012 and Ada 2005. It also briefly revisits a number of problems that were considered for Ada 2005 but rejected for various reasons; the important ones have been solved in Ada 2012.

Finally, it discusses a small number of corrections that have been found necessary since the standard was approved.

Keywords: rationale, Ada 2012.

1 Compatibility

There are two main sorts of problems regarding compatibility. These are termed Incompatibilities and Inconsistencies.

An incompatibility is a situation where a legal Ada 2005 program is illegal in Ada 2012. These can be annoying but not a disaster since the compiler automatically detects such situations.

An inconsistency is where a legal Ada 2005 program is also a legal Ada 2012 program but might have a different effect at execution time. These can in principle be really nasty but typically the program is actually wrong anyway (in the sense that it does not do what the programmer intended) or its behaviour depends upon the raising of a predefined exception (which is generally considered poor style) or the situation is extremely unlikely to occur.

As mentioned below in Section 2, during the development of Ada 2012 a number of corrections were made to Ada 2005 and these resulted in some incompatibilities and inconsistencies with the original Ada 2005 standard. These are not considered to be incompatibilities or inconsistencies between Ada 2005 and Ada 2012 and so are not covered in this section.

1.1 Incompatibilities with Ada 2005

Each incompatibility listed below gives the AI concerned and the paragraph in the AARM which in some cases will give more information. Where relevant, the section in this rationale where the topic is discussed is also given. Where appropriate the incompatibilities are grouped together.

Note that this list only covers those incompatibilities that might reasonably occur. There are a number of others which are so unlikely that they do not seem worth mentioning.

1 – The word **some** is now reserved. Programs using it as an identifier will need to be changed. (AI-176, 2.9)

Adding new reserved words is a very visible incompatibility. Six were added in Ada 95, three in Ada 2005, and now just one in Ada 2012. Perhaps this is the end of the matter. The word **some** is used in quantified expressions; it already was reserved in SPARK [1] where it is used in quantified expressions in proof contexts.

2 – If a predefined package has additional entities then incompatibilities can arise. Thus suppose the predefined package Ada.Stuff has an additional entity More added to it. Then if an Ada 2005 program has a package P containing an entity More then a program with a use clause for both Ada.Stuff and P will become illegal in Ada 2012 because the reference to More will become ambiguous. This also applies if further overloadings of an existing entity are added.

This can be overcome by adding child packages of course. However, adding lots of child packages can be an inconvenience for the user and so in many cases extending a package seemed more appropriate especially if the identifiers concerned are unlikely to have been used by programmers.

The following packages have been extended with additional entities as listed.

Ada.Characters.Handling – Is_Line_Terminator, Is_Mark, Is_Other_Format, Is_Punctuation_Connector, Is_Space. (AI-185, A.3.2)

Ada.Containers – Capacity_Error. (AI-1, A.18.1)

Ada.Containers.Vectors – Assign, Copy, Constant_Reference, Constant_Reference_Type, Iterate, Reference, Reference_Type, Vector_Iterator_Interfaces. (AI-1, AI-212, A.18.2)

There are similar additions to the other containers Ada.Containers.Doubly_Linked_Lists etc.

Ada.Directories – Name_Case_Kind, Name_Case_Equivalence. (AI-49, A.16)

Ada.Dispatching – Yield. (AI-166, D.2.1)

Ada.Environment_Variables – Value. (AI-285, A.17)

Ada.Execution_Time – Interrupt_Clocks_Supported, Separate_Interrupt_Clocks_Supported, Clocks_For_Interrupts. (AI-170, D.14)

Ada.Task_Identification – Environment_Task, Activation_Is_Complete. (AI-189, C.7.1)

Ada.Strings.Fixed – Find_Token. (AI-31, A.4.3)

Ada.Strings.Bounded – Find_Token. (AI-31, A.4.4)

Ada.Strings.Unbounded – Find_Token. (AI-31, A.4.5)

There are similar additions to Ada.Strings.Wide_Fixed, Ada.Strings.Wide_Bounded and Ada.Strings.Wide_Unbounded. (AI-31, A.4.7)

Ada.Tags – Is_Abstract. (AI-173, 3.9)

It seems unlikely that existing programs will be affected by these potential incompatibilities.

3 – Membership tests are no longer allowed as a discrete choice. This is explained in detail in Section 6 of the paper on Expressions. (AI-158, 3.8.1)

4 – Allowing functions to have parameters of all modes led to the introduction of stricter rules on aliasing. It is possible that a program that seemed to work in Ada 2005 is illegal in Ada 2012. See Section 2 of the paper on Structure and Visibility. (AI-144, 6.4.1)

5 – Implicit conversion is now allowed from anonymous access types to general access types. Such conversions can make calls ambiguous in the presence of overloading where only one call was permitted in Ada 2005. Consider

```
type RT is access all T;
function F return RT;
function F return access T;

procedure B(R: RT);
```

and then the call

```
B(F);                    -- ambiguous in Ada 2012
```

The call of B is ambiguous in Ada 2012 because the call could be to either function F. But in Ada 2005, the implicit conversion is not possible and so the call has to be to the first function F. (AI-149, 8.6)

6 – It is now illegal to declare a formal abstract subprogram whose controlling type is incomplete. This is related to various improvements to incomplete types described in Section 3 of the paper on Structure and Visibility. (AI-296, 12.6)

7 – The pragma Controlled has been removed from the language. It was never implemented anyway. (AI-229, 13.11.3)

8 – The package Ada.Dispatching was Pure in Ada 2005 but has been downgraded to Preelaborable because of the addition of Yield. This is unlikely to be a problem. (AI-166, D.2.1)

1.2 Inconsistencies with Ada 2005

Note that this list only covers those inconsistencies that might reasonably occur. There are a number of others which are so unlikely that they do not seem worth mentioning.

1 – The definition of character sets can change with time. It is thus possible that the result of character classification functions for obscure characters might be or become inconsistent. (AI-91, AI-227, AI-266, 2.1, 2.3)

2 – User defined untagged record equality is now defined to compose and be used in generics. Code which assumes that predefined equality reemerges in generics and in predefined equals for composite types could fail. However, it is more likely that this change will fix bugs. (AI-123, 4.5.2)

3 – A stand alone object of an anonymous access type now has dynamic accessibility. This is most likely to make illegal programs now legal. However, it is possible that a program that raised Program_Error in Ada 2005 will not do so in Ada 2012. It seems very unlikely that a program would rely on the raising of this exception. (AI-148, 4.6)

4 – There is an obscure interaction between the change to the composability of equality and renaming. Renaming of user-defined untagged record equality is now defined to call the overridden body so long as the overriding occurred before the renames. Consider

```
package P is
  type T is
    record
      ...
    end record;
                    -- (1) consider renaming here
private
  function "=" (L, R: T) return Boolean;
end P;

with P;
package Q is
  function Equals renames P."=";
end Q;
```

In Ada 2005, Equals refers to the predefined equality, whereas in Ada 2012 it refers to the overridden user-defined equality in the private part. This is so that composed equality and explicit calls on "=" give the same answer. However, if the renaming had been at the point (1) then calling Equal would call the predefined equality. Remember that renaming squirrels away the operation so that it can be retrieved. (AI-123, 8.5.4)

5 – A group budget is now defined to work on a single processor. However, it is unlikely that any implementation of Ada 2005 managed to implement this on multiprocessors anyway. (AI-169, D. 14.2)

2 Retrospective changes to Ada 2005

In the course of the development of Ada 2012, a number of small changes were deemed to apply also to Ada 2005 and thus were classified as binding interpretations rather than amendments. Some were mentioned in previous papers (including that which ensured that package Ada is legal); see Sections 2 and 6 of the paper on Iterators, Pools etc. Most of these do not introduce incompatibilities or inconsistencies so will not be discussed further.

A few binding interpretations do introduce minor incompatibilities or inconsistencies and will now be briefly discussed.

2.1 Incompatibilities with original Ada 2005

There are a small number of incompatibilities between the original Ada 2005 and that resulting from various corrections.

1 – The rules for full conformance have been strengthened; for example, null exclusions must now match. (AI-46, AI-134, AI-207, 6.3.1)

2 – When an inherited subprogram is implemented by a protected function, the first parameter has to be an **in** parameter, but not an access to variable type. Ada 2005 allowed access to variable parameters in this case; the parameter will need to be changed to access to constant by the addition of the **constant** keyword. (AI-291, 9.4)

3 – A missing rule is added that a limited with clause cannot name an ancestor unit. (AI-40, 10.1.2)

4 – Matching of formal access to subprogram types uses subtype conformance in Ada 2012 whereas it only used mode conformance in original Ada 2005. This change was necessary to avoid undefined behaviour in some situations. (AI-288, 12.5.4)

5 – An address attribute with a prefix of a subprogram with convention Intrinsic is now illegal. This is discussed in Section 6 of the paper on Iterators, Pools etc. (AI-95, 13.3)

6 – Stream attributes must be specified by a static subprogram name rather than by a dynamic expression. (AI-39, 13.13.2)

7 – The use of discriminants on Unchecked_Union types is now illegal in record representation clauses. It makes no sense to specify the position of something that is not supposed to exist. (AI-26, B.3.3)

8 – A nonvolatile generic formal derived type precludes a volatile actual type. (AI-218, C.6)

9 – The restriction No_Relative_Delay has been extended to also prohibit a call of Timing_Events.Set_Handler with a Time_Span parameter. (AI-211, D.7)

10 – Various restrictions have been reworded to prevent the bypassing of the restriction by calling the forbidden subprogram via renames. (AI-211, D.7)

2.2 Inconsistencies with original Ada 2005

There are a small number of inconsistencies between the original Ada 2005 and that resulting from various corrections.

1 – The description of Dependent_Tag has been changed to say that it must raise Tag_Error if there is more than one type that matches the requirements. (AI-113, 3.9)

2 – A curious omission regarding checking arrays allows a component in an aggregate whose value is given as <> even if the component is outside the bounds. It is now clarified that Constraint_Error is raised. (AI-37, 4.3.3)

3 – The first procedure Split in Ada.Calendar.Formatting raises Time_Error for a value of exactly 86400.0. This was unspecified in Ada 2005. (AI-238, 9.6.1)

4 – An address attribute with a prefix of a generic formal subprogram whose actual parameter has convention Intrinsic now raises Program_Error. (AI-95, 13.3)

5 – User specified external tags that conflict with other external tags now raise Program_Error or are illegal. (AI-113, 13.3)

6 – The definition of Set_Line is corrected. As originally defined in Ada 95 and Ada 2005, Set_Line(1) could call New_Line(0) which would raise Constraint_Error which is unhelpful. This was mentioned right at the end of the Postscript in the Rationale for Ada 2005 [2]. (AI-38, A.10.5)

7 – The definitions of Start_Search, Search, Delete_Directory, and Rename are clarified so that they raise the correct exception if misused. (AI-231, A.16)

8 – If Count = 0 for a container Insert subprogram that has a Position parameter, the Position parameter is set to the value of the Before parameter by the call. The original wording remained silent on this. (AI-257, A.18.3)

3 Unfinished topics from Ada 2005

A number of topics which seemed to be good ideas initially were abandoned during the development of Ada 2005 for various reasons. Usually the reason was simply that a good solution could not be produced in the time available and the trouble with a bad solution is that it is hard to put it right later. This section briefly reconsiders these topics which were discussed in the Rationale for Ada 2005 [2]; some have now been solved in Ada 2012; the others were considered unimportant.

3.1 Aggregates for private types

The <> notation was introduced in Ada 2005 for aggregates to mean the default value if any. A curiosity is that we can write

```
type Secret is private;

type Visible is
  record
    A: Integer;
    S: Secret;
  end record;

X: Visible := (A => 77; S => <>);
```

but we cannot write

```
S: Secret := <>;                    -- illegal
```

The argument is that this would be of little use since the components take their default values anyway.

For uniformity it was proposed that we might allow

```
S: Secret := (others => <>);
```

for private types and also for task and protected types. One advantage would be that we could then write

```
S: constant Secret := (others => <>);
```

whereas it is not possible to declare a constant of a private type because we are unable to give an initial value.

However, discussion of this issue led into a quagmire in Ada 2005 and so was abandoned. It remains abandoned in Ada 2012!

3.2 Partial generic instantiation

Certain attempts to use signature packages led to circularities in Ada 95. Consider

```
generic
  type Element is private;
  type Set is private;
  with function Union(L, R: Set) return Set  is <>;
  with function Intersection(L, R: Set) return Set is <>;
  ... -- and so on
package Set_Signature is end;
```

Remember that a signature is a generic package consisting only of a specification. When we instantiate it, the effect is to assert that the actual parameters are consistent and the instantiation provides a name to refer to them as a group.

If we now attempt to write

```
generic
  type Elem is private;
  with function Hash(E: Elem) return Integer;
package Hashed_Sets is
  type Set is private;
  function Union(L, R: Set) return Set;
  function Intersection(L, R: Set) return Set;
  ...
  package Signature is new Set_Signature(Elem, Set);
private
  type Set is
    record
      ...
    end record;
end Hashed_Sets;
```

then we are in trouble. The problem is that the instantiation of Set_Signature tries to freeze the type Set prematurely.

After a number of false starts this problem is partially overcome in Ada 2012 by the introduction of incomplete formal generic parameters. This is discussed in Section 3 of the paper on Structure and Visibility. See also Section 4.1 of this paper.

3.3 Support for IEEE 559: 1989

The proposal was to provide full support for all aspects of IEEE 559 arithmetic such as NaNs (a NaN is Not A Number). This would have necessitated adding attributes such as S'Infinity, S'Is_NaN, S'Finite and so on plus a package Ada.Numerics.IEC_559.

The proposal was abandoned because it would have had a big impact on implementers and it was not clear that there was sufficient demand. It was not reconsidered for Ada 2012.

3.4 Defaults for generic parameters

Generic subprogram parameters and object parameters of mode in can have defaults. But other parameters such as packages and types cannot. This was considered irksome and untidy and efforts were made to define a suitable notation for all possible generic parameters.

However, it was abandoned partly because an appropriate syntax seemed hard to find and more importantly, it was not felt to be that important. Again, it was not deemed important enough to be reconsidered for Ada 2012.

3.5 Pre/post-conditions for subprograms

The original proposal was to add pragmas such as Pre_Assert and Post_Assert. Thus in the case of a subprogram Push on a type Stack we might write

```
procedure Push(S: in out Stack; X: in Item);
pragma Pre_Assert(Push, not Is_Full(S));
pragma Post_Assert(Push, not Is_Empty(S));
```

This was all abandoned in Ada 2005 for various reasons; one being that pragmas are ugly for such an important matter.

However, this is neatly solved in Ada 2012 by the introduction of aspect specifications so we can now write

```
procedure Push(S: in out Stack; X: in Item)
  with
    Pre => not Is_Full(S),
    Post => not Is_Empty(S);
```

which is really excellent; this is discussed in detail in the paper on Contracts and Aspects.

3.6 Type and package invariants

This defined further pragmas similar to those in the previous proposal but concerned with packages and types. Thus the pragma Package_Invariant proposed for Ada 2005 identified a function returning a Boolean result. This function would be implicitly called after the call of each subprogram in the package and if the result were false the behaviour would be as for an Assert pragma that failed.

This proposal was also abandoned for Ada 2005. However, Ada 2012 has introduced type invariants thus

```
type Stack is private
  with Type_Invariant => Is_Unduplicated(Stack);
```

as discussed in the paper on Contracts and Aspects. On the other hand, package invariants remain abandoned.

3.7 Exceptions as types

This proposal originally arose out of a workshop organized by Ada-Europe. It was quite complex and considered far too radical a change and probably expensive to implement. As a consequence it was slimmed down considerably. But having been slimmed down it seemed pointless and was then abandoned. The only part to survive was the idea of raise with message which became a separate AI and was incorporated into Ada 2005.

This was not pursued in Ada 2012.

3.8 Sockets operations

This seemed a very good idea at the time but no detailed proposal was forthcoming and so it died. It has been left dead.

3.9 In out parameters for functions

The proposal was to allow functions to have parameters of all modes. The rationale for the proposal was well summarized thus "Ada functions can have arbitrary side effects, but are not allowed to announce that in their specifications".

But strangely, this AI was abandoned quite early in the Ada 2005 revision process on the grounds that it was "too late". (Perhaps too late in this context meant 25 years too late.)

However, in Ada 2012, the bullet has been bitten and functions can indeed now have parameters of all modes. See the discussion in Section 2 of the paper on Structure and Visibility.

3.10 Application defined scheduling

The International Real-Time Ada Workshops have been a source of suggestions for improvements to Ada. The Workshop at Oporto suggested a number of further scheduling algorithms [3]. Most of these such as Round Robin and EDF were included in Ada 2005. But that for application defined scheduling was not.

No further action on this topic was taken in Ada 2012.

4 Unfinished topics for Ada 2012

A number of topics which seemed to be good ideas initially were abandoned during the development of Ada 2012 for various reasons. It is interesting to note that there are far fewer of these loose ends than there were in Ada 2005. The following deserve mention.

4.1 Integrated packages (AI-135)

Difficulties sometimes arise with nested packages. Consider for example a package that needs to export a private type T and a container instantiated for that type. We cannot write

```
package P is
  type T is private;
  package T_Set is new Ordered_Sets(T);
private
  ...
end P;
```

because the type T is not frozen. We have to write something like

```
package P is
  package Inner is
    type T is private;
  private
    ...
  end Inner;
  package T_Set is new Ordered_Sets(Inner.T);
end P;
```

What we now want is some way to say that the declarations in Inner are really at the level of P itself after all. In other words we want to integrate the package Inner with the outer package P.

Various attempts were made to solve this by another kind of use clause or perhaps by putting Inner in a <> box. But all attempts led to difficulties so this remains unresolved.

4.2 Cyclic fixed point (AI-175)

Measurements in the physical world of Euclid and Newton are either lengths or angles. Angles are cyclic in nature and so can be mapped with a modular type. However, this leaves scaling in the hands of the user and is machine dependent. Consideration was given to the possibility of a cyclic form of fixed point. Sadly, there was much hidden complexity and so no solution was agreed.

One might have thought that it would be easy to use the natural wrap-around hardware. However, with a binary machine, if 180 degrees is held exactly then 60 degrees is not which excludes an exact representation of an equilateral triangle. The whole point about using fixed point is that it is precise but it just doesn't work unless the hardware uses a base with divisibility by 60. The Babylonians would have understood. The text of AI-175 includes a generic which might be useful for many applications.

4.3 Global annotations (AI-186)

The idea here was that the specification of a subprogram should have annotations indicating the global objects that it might manipulate. For example a function can have side effects on global variables but this important matter is not mentioned in the specification. This topic has strong synergy with the information given in contracts such as pre- and postconditions. However, it was abandoned perhaps because of the complexity arising from the richness of the full Ada language. It should be noted that such annotations have always featured in SPARK as comments and moreover, at the time of writing, are being considered using the aspect notation in a new version of SPARK.

4.4 Shorthand for assignments (AI-187)

Consideration was given to having some short of shorthand for assignments where source and target have commonality as in statements such as

```
A(I) := A(I) + 1;
```

But maybe the thought of C++ was too much. In any event no agreement that it was worthwhile was reached and there was certainly no agreement on what syntax might be acceptable.

5 Postscript

It should also be noticed that a few corrections and improvements have been made since Ada 2012 was approved as a standard. The more important of these will now be discussed.

A new form of expression, the raise expression, is added (AI12-22). This means that by analogy with

```
if X < Y then
  Z := +1;
elsif X > Y then
  Z := –1;
else
  raise Error;
end if;
```

we can also write

```
Z := (if X<Y then 1 elsif X>Y then –1 else raise Error);
```

A raise expression is a new form of relation so the syntax for relation (see Section 6 of the paper on Expressions) is extended as follows

```
relation ::=
        simple_expression [relational_operator simple_expression]
      | simple_expression [not] in membership_choice_list
      | raise_expression

raise_expression ::=
        raise exception_name [with string_expression]
```

Since a raise expression is a relation it has the same precedence and so will need to be in parentheses in some contexts. But as illustrated above it does not need parentheses when used in a conditional expression which itself will have parentheses.

Raise expressions will be found useful with pre- and postconditions. Thus if we have

```
procedure Push(S: in out Stack; X: in Item)
  with
    Pre => not Is_Full(S);
```

and the precondition is false then Assertion _Error is raised. But we can now alternatively write

```
procedure Push(S: in out Stack; X: in Item)
  with
    Pre => not Is_Full(S) or else raise Stack_Error;
```

and of course we can also add a message thus

```
    Pre => not Is_Full(S) or else raise Stack_Error with "wretched stack is full";
```

On a closely related topic the new syntax for membership tests (also see Section 6 of the paper on Expressions) has been found to cause ambiguities (AI12-39).

Thus

```
A in B and C
```

could be interpreted as either of the following

```
(A in B) and C                    -- or
A in (B and C)
```

This is cured by changing the syntax for relation yet again to

```
relation ::=
        simple_expression [relational_operator simple_expression]
      | tested_simple_expression [not] in membership_choice_list
      | raise_expression
```

and changing

```
membership_choice ::=
        choice_expression | range | subtype_mark
```

to

```
membership_choice ::=
        choice_simple_expression | range | subtype_mark
```

Thus a membership_choice no longer uses a choice_expression. However, the form choice_expression is still used in discrete_choice.

A curious difficulty has been found in attempting to use the seemingly innocuous package Ada.Locales described in Section 4 of the paper on the Predefined Library.

The types Language_Code and Country_Code were originally declared as

```
type Language_Code is array (1 .. 3) of Character range 'a' .. 'z';

type Country_Code is array (1 .. 2) of Character range 'A' .. 'Z';
```

The problem is that a value of these types is not a string and cannot easily be converted into a string because of the range constraints and so cannot be a simple parameter of a subprogram such as Put. If LC is of type Language_Code then we have to write something tedious such as

```
Put(LC(1));  Put(LC(2));  Put(LC(3));
```

Accordingly, these types are changed so that they are derived from the type String and the constraints on the letters are then imposed by dynamic predicates. So we have

```
type Language_Code is new String(1 .. 3)
  with Dynamic_Predicate => (for all E of Language_Code => E in 'a' .. 'z';
```

with a similar construction for Country_Code (AI12-37).

Readers might like to contemplate whether this is an excellent illustration of some of the new features of Ada 2012 or simply an illustration of static strong or maybe string typing going astray.

AI12-45 notes that pre- and postconditions are allowed on generic units but they are not allowed on instances. See Section 3 of the paper on Contracts and Aspects where this topic should have been mentioned.

Another modification in this area is addressed by AI12-44 which states that type invariants are not checked on **in** parameters of functions but are checked on **in** parameters of procedures. See Section 4 of the paper on Contracts and Aspects. This change was necessary to avoid infinite recursion which would arise if an invariant itself called a function with a parameter of the type. Note also that a class wide invariant could not be used at all without this modification.

A further aspect, Predicate_Failure, is defined by AI12-54-2. The expected type of the expression defined by this aspect is String and gives the message to be associated with a failure. So we can write

```
subtype Open_File_Type is File_Type
  with
     Dynamic_Predicate => Is_Open(Open_File_Type),
     Predicate_Failure =>  "File not open";
```

If the predicate fails then Assertion_Error is raised with the message "File not open". See Section 5 of the paper on Contracts and Aspects.

We can also use a raise expression and thereby ensure that a more appropriate exception is raised. If we write

```
Predicate_Failure => raise Status_Error with "File not open";
```

then Status_Error is raised rather than Assertion_Error with the given message. We could of course explicitly mention Assertion_Error thus by writing

```
Predicate_Failure => raise Assertion_Error with "A message";
```

Finally, we could omit any message and just write

```
Predicate_Failure => raise Status_Error;
```

in which case the message is null.

A related issue is discussed in AI-71. If several predicates apply to a subtype which has been declared by a refined sequence then the predicates are evaluated in the order in which they occur. This is especially important if different exceptions are specified by the use of Predicate_Failure since without this rule the wrong exception might be raised. The same applies to a combination of predicates, null exclusions and old-fashioned subtypes.

This can be illustrated by an extension of the above example. Suppose we have

```
subtype Open_File_Type is File_Type
  with
     Dynamic_Predicate => Is_Open(Open_File_Type),
     Predicate_Failure => raise Status_Error;
```

```
subtype Read_File_Type is Open_File_Type
  with
    Dynamic_Predicate => Mode(Real_File_Type) = In_File,
    Predicate_Failure => raise Mode_Error with "Can't read file: " & Name(Read_File_Type);
```

The subtype Read_File_Type refines Open_File_Type. If the predicate for it were evaluated first and the file was not open then the call of Mode would raise Status_Error which we would not want to happen if we wrote

```
if F in Read_File_Type then ...
```

Care is needed with membership tests. The whole purpose of a membership test (and similarly the Valid attribute) is to find out whether a condition is satisfied. So if we write

```
if X in S then
  ...               -- do this
else
  ...               -- do that
end if;
```

we expect the membership test to be true or false. However, if the evaluation of S itself raises some exception then the purpose of the test is violated.

It is important to understand these related topics. Another example might clarify. Suppose we have a very simple predicate as in Section 5 of the paper on Contracts and Aspects such as

```
subtype Winter is Month
  with Static_Predicate => Winter in Dec | Jan | Feb;
```

where

```
type Month is (Jan, Feb, Mar, Apr, ..., Nov, Dec);
```

and we declare a variable W thus

```
W: Winter := Jan;
```

If we now do

```
W := Mar;
```

then Assertion_Error will be raised because the value Mar is not within the subtype Winter (we assume that the assertion policy is Check). If, however, we would rather have Constraint_Error raised then we can modify the declaration of Winter to

```
subtype Winter is Month
  with Static_Predicate => Winter in Dec | Jan | Feb,
       Predicate_Failure => raise Constraint_Error;
```

and then obeying

```
W := Mar;
```

will raise Constraint_Error.

On the other hand suppose we declare a variable M thus

```
M: Month := Mar;
```

and then do a membership test

```
if M in Winter then
  ...               -- do this if M is a winter month
```

```
else
   ...                    -- do this if M is not a winter month
end if;
```

then of course no exception is raised since this is a membership test and not a predicate check.

Note however, that we could write something odd such as

```
subtype Winter2 is Month
   with Dynamic_Predicate => (if Winter2 in Dec | Jan | Feb then true else raise E);
```

then the very evaluation of the predicate might raise the exception E so that

```
M in Winter2
```

will either be true or raise the exception E but will never be false. Note that in this silly example the predicate has to be a dynamic one because a static predicate cannot include a raise expression.

So this should clarify the reasons for introducing Predicate_Failure. It enables us to give a different behaviour for when the predicate is used in a membership test as opposed to when it is used in a check and it also allows us to add a message.

Finally, it should be noted that the predicate expression might involve the evaluation of some subexpression perhaps through the call of some function. We might have a predicate describing those months that have 30 days thus

```
subtype Month30 is Month
   with Static_Predicate => Month30 in Sep | Apr | Jun | Nov;
```

which mimics the order in the nursery rhyme. However, suppose we decide to declare a function Days30 to do the check so that the subtype becomes

```
subtype Month30 is Month
   with Dynamic_Predicate => Days30(Month30);
```

and for some silly reason we code the function incorrectly so that it raises an exception (perhaps it accidentally runs into its **end** and always raises Program_Error). In this situation if we write

```
M in Month30
```

then we will indeed get Program_Error and not false.

Perhaps this whole topic can be summarized by simply saying that a membership test is not a check. Indeed a membership test is often useful in ensuring that a subsequent check will not fail as was discussed in Section 4 of the paper on Iterators, Pools etc.

On a rather different topic, AI12-28 discusses the import of variadic C functions (that is functions with a variable number of parameters). In Ada 95, it was expected that such functions would use the same calling conventions as normal C functions; however, that is not true for some targets today. Accordingly, this AI adds additional conventions to describe variadic C functions so that the Ada compiler can compile the correct calling sequence.

Finally, an important modification is made to the topic of dispatching domains by AI12-33. See Section 3 of the paper on Tasking and Real-Time.

As defined originally, a dispatching domain consists of a set of processors whose CPU values are contiguous. However, this is unrealistic since CPUs are often grouped together in other ways. Accordingly, the package System.Multiprocessors.Dispatching_Domains is extended by the addition of a type CPU_Set and two further functions thus

```
type CPU_Set is array (CPU range <>) of Boolean;
```

```
function Create(Set: CPU_Set) return Dispatching_Domain;
function Get_CPU_Set(Domain: Dispatching_Domain) return CPU_Set;
```

So if we want to create a domain consisting of processors 0, 4, and 8 we can write

```
My_Set: CPU_Set(0 .. 8) := (0 | 4 | 8 => true, others => false);
```

and then

```
My_Domain: Dispatching_Domain := Create(My_Set);
```

and so on. The function Get_CPU_Set can be applied to any domain and returns the appropriate array representing the set of CPUs. Note that this function can be applied to any domain and not just to one created from a CPU_Set.

6 Acknowledgements

This is the last of the papers in this series and so this seems a good moment to once more thank Randy Brukardt for his diligence and patience in reviewing various drafts and putting me back on track when I got lost.

I must also thank AdaCore and the British Standards Institute for financial support for attending various meetings.

As usual, writing this rationale has been a learning experience for me and I trust that readers will also have found the material useful in learning about Ada 2012. An integrated description of Ada 2012 as a whole will be found in a forthcoming version of a familiar textbook.

References

1. John Barnes (2012) *SPARK – The proven approach to High Integrity Software,* Altran Praxis.
2. John Barnes (2008) *Ada 2005 Rationale*, LNCS 5020, Springer-Verlag.
3. ACM (2003) *Proceedings of the 12th International Real-Time Ada Workshop*, Ada Letters, Vol 23, No 4.