Final Minutes of 12th ARG Meeting

17-19 November 2000
Columbia MD

Attendees: John Barnes, Randy Brukardt, Gary Dismukes, Bob Duff, Kiyoshi Ishihata (Friday only), Mike Kamrad, Pascal Leroy, Stephan Michell, Erhard Ploedereder, Tucker Taft, Joyce Tokar
Observers: Alan Burns (of University of York, UK), Mike Yoder (of Top Layer Networks ,USA)

Meeting Summary

The meeting convened on 17 November 00 at 13:00 hours at the Sheraton Hotel in Columbia MD and adjourned at 13:00 hours on 19 November 00.

As advertised in the draft agenda for the meeting, the overwhelming emphasis of the meeting was the review of the submitted amendments, with only a few non-amendments AIs receiving review. All outstanding action items were reviewed and reassigned. A list of potential targets for amendments was created and a large subset of them were assigned for AIs.

Once again while there was no official vote, I (Mike) think that I speak for others in thanking Erhard for facilitating the excellent arrangements at the hotel.

Next Meeting

The next meeting is set for 18-20 May 2001 in Leuven, Belgium, in conjunction with the next Ada-Europe Conference. The ARG meeting will begin in the early afternoon of 18 May, after the WG9 meeting has adjourned.

Old Action Items

The old action items were reviewed. Action items that were assigned to former members were either reassigned or were dramatically lowered in priority (See section on Revised ARG Procedures for explanation of the priority change); the decision is shown in square brackets by either the name of the person to whom the action was reassigned, or the status otherwise.

Randy Brukardt: new AIs on Unsuppress [done] and `Object_size; [remains]
Norm Cohen: AI-133 [Tuck], AI-167 [very low priority], AI-173 [very low priority]
Robert Dewar: AI-85 (distribute an “append" Test to comp.lang.ada and collate results to AI) [Randy]
Gary Dismukes: AI-158 [definitely an amendment; is it still relevant?], AI-196 [to be completed]
Robert Eachus: AI-100 [very low priority], AI-172 [very low priority], AI-174 [Randy: merge with 195 or real], AI-185 [Mike Y], AI-186 [Pascal]
Mike Kamrad: new AI on Assert pragma [to be completed]
Stephen Michell: AI-148 [to be completed], AI-187 [very low priority]
Erhard Ploedereder: AI-209 [done (and submitted to HRG)], AI-212 [done]
Tucker Taft: AI-162 [to be completed], AI-188 [to be completed], AI-189 [no action], AI-191 [very low priority], AI-195[done], AI-216 [done], AI-214 [to be completed]

All: Create tests for assigned AIs [ongoing]
Bob Duff: Be the test creator of last resort [ongoing], distribute Design Note on "inward closures" [done]
Randy Brukardt: distribute ,inward-closures" study from prototyping Project [done]

New Action Items

The combined old and new actions items from the meeting are shown below:

John Barnes:
- Downward closures
- AI-241

Randy Brukardt:
- AI-85

- AI-174
- Post existing test objectives and assignment to the Grab Bag on the ARG web site;
- Publish proposed test objectives to ARG
- AI-218
- AI-224
- Standard library enhancement for directories
- Object_size attribute
- AI-227
- AI-235
- AI-236
- AI-237

Gary Dismukes:
- AI-158
- AI-196

Bob Duff:
- Be the test creator of last resort
- Fix limited types a bit to permit aggregates and initialization by function call [jointly with Tuck]

Mike Kamrad:
- Amendment on Assert pragma
- Various items to be standardized [jointly with Mike Yoder]
  - Discard_name & ‘image
  - External_tag
  - Storage_IO of tagged types
  - Array indexed by holey enumeration
  - Static elaboration
  - GNAT attributes and pragmas
- CPU time (separate from real-time) [jointly with Joyce]

Pascal Leroy:
- AI-186
- AI-230 [jointly with Tuck]
- Exception hierarchy
- Raise exception with objects

Steve Michell:
- AI-148
- Tagged protected types

Tucker Taft:
- AI-133
- AI-162
- AI-188
- Tuck will assume the creation of a mission statement for the next meeting and he will take contributions from other ARG members
- AI-195
- AI-214
- AI-216
- AI-217
- AI-230 [jointly with Pascal]
- AI-231
- Object’operator
- Interfaces and Multiple Inheritance
- More stream representation control
- User defined assignment/subtype conversion
- Physical units (length/dimensional analysis), whereby square inch units are generated by result of multiplying inches; subtypes with special attributes would be used to provide the specifications of these dimensions, reducing the introductions of lots of extra operators.
- Fix limited types a bit to permit aggregates and initialization by function call [jointly with Bob]

Joyce Tokar:
- Ravenscar Profile [Joyce]
- CPU time (separate from real-time) [jointly with Mike Kamrad]

Mike Yoder:
- AI-185
- Various items to be standardized [jointly with Mike Kamrad]
  - Discard_name & 'image
  - External_tag
  - Storage_IO of tagged types
  - Array indexed by holey enumeration
  - Static elaboration
  - GNAT attributes and pragmas

All:
- Create tests for assigned AIs [still outstanding]
- Review proposed test objectives and vote on these objectives
- Recommend new members

Revised ARG Procedures

Erhard opened the discussion on this topic by recommending that future ARG activity/focus should be with an 80/20 ratio for amendment AIs vs. traditional AIs. This may require a change to ARG procedures where every submitted comment gets an AI and is put onto the agenda. He wants to filter them before they reach ARG attention. Randy feels these should see some daylight. Tuck suggests that the filtered stuff gets "no action" status. Erhard doesn't want them to get to AI status at all; instead he wants them placed into a new category of "no action" commentary, named Ada Commentary. Approved 11-0-0.

On the issue of formally announcing the change of emphasis on amendment AIs, the group asked whether a call for amendments or even the formal announcement of these amendments would be needed? The risk is inundating the ARG with many language suggestions. Most thought the community will find out on its own and therefore it wasn't necessary to make this announcement to them. Naturally, we'll accept amendment proposals arriving via ada-comment. What about formally informing vendors? Most of them seem to be on the ARG list already. It appears that they all know where to find the AIs so there is no need to formally announce to them anyway. Hence, no action here.

On the issue of test objectives and test developments for approved AIs, Randy has produced a draft of the objectives and maintains the assignments of test development for the currently approved AIs. Unfortunately the ARG doesn't know them; hence, he will post them onto Grab Bag at the normal web site and also e-mail them to ARG. This lead to the question of who should review the objectives to determine whether they should be tested? On a straw vote, it was decided that this review should be done by the ARG. The procedure for this review will be as follows:

1. Randy will publish the proposed objectives.
2. A review of four weeks will precede a letter ballot.
3. Approval is by 2/3s of ARG voting and voting is done per objective.

All of this will be announced to the community, in particular the vendor community.

On the issue of ARG membership, Erhard is concerned that we are reaching the low end of critical mass on active ARG membership; he would like to see the active membership be increased to fifteen or more members. Consequently he strongly encouraged ARG members to suggest names for new members. Immediately, Tuck recommended Mike Yoder as a new member.

Directions for Amendments

This is a summary of two different discussions on Friday and Saturday. Tuck came to the meeting with some ideas on directions for new amendments, largely from his presentation to SIGAda on Thursday afternoon. He recommends that the following four criteria be emphasized for amendments:
- Safety of applications written in Ada, which continues to be Ada's strong suit
- Portability, with emphasis on APIs to existing capabilities

- Interoperability with other languages and systems
- Accessibility and ease of transition from idioms in other programming and modeling languages, such as UML to Ada

Proposed amendments should be restricted to those that improve marketability of the language and programmer productivity. He further recommends that the minimum criteria for submitting amendments must include an example of use and a conformance test. Tuck will assume the creation of a mission statement for the next meeting and he will take contributions from other ARG members.

To avoid overloading the ARG, Erhard stated that the ARG should not be the source of APIs. The appropriate vehicle for these APIs are secondary standards, which will expedite their approval. Furthermore secondary standards are easier for other groups outside of the ARG to produce. Presumably, the WG9 would provide direction and management of secondary standard development.

Erhard recommended that an effort be made to officially solicit proposals for these packages. The targets for this solicitation could be

- Comp.lang.ada
- Team-ada
- Ada Letters
- SIGAda/other Ada groups
- Vendors, ARA

There was a specific recommendation to ask Clyde Roby to organize this effort with official status from ARG. Proposals need to be complete and well supported.

On the next day, the members did discuss some specific targets for amendments and produced this unofficial list of potential amendments. The order of the list is not significant. The names in brackets are ARG members who volunteered to sponsor this particular amendment:

- Ravenscar Profile [Joyce]
- Object'operator [Tuck]
- Interfaces and multiple inheritance [Tuck]
- Standard library enhancement for directories [Randy]
- Untagged record primitive operations (composability and reemergence in generics)
- Tagged protected types [Steve]
- More stream representation control [Tuck]
- User defined assignment/subtype conversion [Tuck]
- Exception hierarchy [Pascal]
- Raise exception with objects [Pascal]
- Fix limited types a bit to permit aggregates and initialization by function call [Bob & Tuck]
- Physical units (length/dimensional analysis), whereby square inch units are generated by result of multiplying inches; subtypes with special attributes would be used to provide the specifications of these dimensions, reducing the introductions of lots of extra operators. [Tuck]
- Pure access types, so that packages with these access types may be declared to be Pure.
- Extensible enumeration types (which was given a low priority at the meeting)
- Various items to be standardized [The Mikes]
    - Discard_name & 'image
    - External_tag
    - Storage_IO of tagged types
    - Array indexed by holey enumeration
    - Static elaboration
    - GNAT attributes and pragmas (there was a previous round at the Marlboro ARG meeting; 'Object_Size is the only open proposal remaining)
- CPU time (separate from real-time) [Joyce, Mike Kamrad]
- 1 .. <> arrays, namely the declaration of arrays with an unconstrained upper bound and the lower bound fixed.
- C++ interface
- Untagged 'Class (another low priority)

- Others => <> in the specification of generic formal packages to allow partially constrained instantiations
- Downward closures [John Barnes]
- SPARKish/Eiffel-like invariants/pre&post conditions

Several of these proposed amendments were discussed towards the end of the meeting.

Detailed Review

The minutes for the detailed review of AIs are divided into existing amendment AIs, proposed targets for amendment AIs and non-amendment AIs.

Detailed Review of Amendment AIs

AI-216: Unchecked Unions -- Variant Records Without Run-Time Discriminant

Tuck explained the significant changes he made to the AI. The first was to add more details and permissions regarding type conversions to and from a derived type that also has the pragma Unchecked_Union applied to it. This change was warmly received. The second change was to permit the specification of the discriminant of a type that has the pragma Unchecked_Union applied to it in a component clause of a record representation clause. This change was not received very well, as it appears to run counter to the goal of eliminating storage for the discriminants of these types. All the wording for the latter change was dropped completely from the AI, including:

- The second parenthetical phase in the first paragraph (of the proposal section)
- The second parenthetical phase in the fourth paragraph

The wording was modified as follows:

- Replace the last sentence of the fifth paragraph with this sentence
  "Note that the discriminant may be used to govern a variant part or as a default initial value for a component."

- Replace the current last paragraph with this paragraph:
  "Record representation clauses are permitted for unchecked unions. No space is given for a discriminant; it is illegal to mention a discriminant explicitly in a component clause."

Tuck did such a good job of handling type conversions that Pascal suggested that some implementations may wish to allow this pragma for tagged types. At first glance, this seems to be worthwhile as an implementation-defined feature but when this AI is applied to tagged types as well, the wording of the AI becomes very complicated. Therefore, it was decided that the AI is explicitly targeted at untagged types and that permission for applying it to tagged types is implementation-defined with implementation-defined semantics. This following paragraph is placed at the end of the AI proposal:

> "An implementation may support this pragma for tagged record types or record extensions, with implementation-defined semantics."

Questions were raised on the convention for the type, in particular on overriding the implied default convention C. After some discussion it was decided that it is to be implementation-defined which, if any, conventions can override convention C in these cases.

There were minor changes that were recommended:

- Remove the Note at the end of the third paragraph as it adds no value; there should not be a component clause for the discriminant.
- Drop the phrase "its ultimate ancestor" in the first sentence of the eighth paragraph on type conversions.
- Replace the use of "(s)" with just "s".

Vote on intent of proposal, with changes, 10-0-0. Tuck will supply the wording in the next revision.

AI-217: Handling Mutually Dependent Type Definitions that Cross Packages

There were two points argued on the Completeness Check Issue of this AI:

- the presence of the specified library package in the compilation of the compilation unit that refers to the library package in a with_type_clause
- the specification of when the incomplete type is checked against the full type

On the first point, it has been argued that the specified library package (in a with_type_clause) does not need to exist for the referring compilation unit to compile, the model being that no semantic dependency on the specified library package can be created (or else the description of the semantics become much more complicated and need to be checked for ramifications, e.g., on elaboration ordering rules). Unfortunately, GNAT has made the presence of the specified library package mandatory in its implementation, making this issue more than an intellectual exercise. One approach is to exempt implementations from this rule by simply not creating a conformance test for it. But this loop hole can't be guaranteed since the results of the review of test objectives is not known nor can the future of the ACATS be foreseen. A straw vote, with Gary dissenting, was in favor of the rules as written by Tuck. It was noted that full conformance to an amendment could only be required after ISO approval, i.e., optimistically after 2006, by which time attitudes may have changed anyway.

On the second point about when the completeness check is made, Tuck had made the following proposal in an email earlier this year (which is found in the appendix of the AI):

> "There seems to be some reasonable agreement on these 4 AIs, which is good. However, we definitely are still debating the issue of checking the incomplete type against the full type. Here is one idea:
>
> *Require* the check only when the incomplete type "meets" the full type in a particular usage.
>
> There are few places where the incomplete type can be used:1) As the designated type in an access type definition 2) As a parameter type in an access-to-subp definition 3) As a parameter type (and result?) in a subprogram declaration (if incomplete tagged).
>
> These "meet" the full type when
> 1) The access type is dereferenced.
> 2) '[Unchecked_]Access or "new" is used to create a value of the access type.
> 3) The subprogram is called.
> 4) The subprogram (proper) body is provided.
>
> In these four cases, at a minimum, the implementation must verify that the full type exists in the specified package, is a first subtype, and is tagged if the incomplete type is tagged.
>
> Implementations are permitted to verify these requirements in any unit that has a semantic dependence on both the comp[ilation] unit defining the incomplete type *[by means of the with_type_clause]* and the one defining the full type.
>
> The primary wording in the RM could be kept simple, simply requiring that they be the same. An implementation permission (or requirement) could indicate the places where the rule needs to be verified."

It was suggested that this wording could be incorporated in the AARM as guidance in implementation notes on this feature.

At the meeting Tuck suggested that these checks be required at post-compilation (for safety sake) and that the check may never be made if the types never meet. And he suggested that the second paragraph of the proposal on changes to RM 10.1.2 be modified with a change to the first sentence (shown below) and his wording from the email be made part of post-compilations rules.

> "If a partition includes a library package as well as a compilation unit that specifies the library package in a with_type_clause, a type declaration (not just a subtype declaration) for the named type shall appear immediately within the visible part of the package. In addition, if the package is a descendant of a private child of some ancestor package, then the unit with the with_type_clause shall also be a descendant of a private child of that same ancestor package."

Do we permit a loop hole (by omitting tests) for implementations that would find this check onerous? This is the same situation as for the previous point. For implementations that already do post-compilations checks, this shouldn't be too onerous.

On the other open issue on extending the permission to let generics have with_type_clauses, there appeared to be no specific reason why such a restriction should be imposed.

Gary objected to the permission for one of these incomplete (tagged) types to be the result type of a function (because a value of the type would need to be created). There was a discussion of the function calls in default expressions, which does not cause problems per se; as always, no such call may be executed in a context, in which the full type isn't available. The group concluded that calls on functions with results of such incomplete types cannot and should not be permitted.

There was a recommendation to tighten the wording on the presence of the full type in the specified library in the first paragraph of the proposed changes to RM 10.1.2 as follows:

> "The full type definition for the type is presumed to *be declared immediately* in the specified *library* package, but no semantic dependence is created on this package, and the package need not be present in the environment at the time the with_type_clause is processed."

The directed changes on this AI were approved, 7-1-3, the dissenting vote being on the question of the existence of the package at compilation time. Tuck will revise the AI.

AI-218: Accidental overloading when overriding

After reviewing the reason why the pragma approach is better than a keyword approach, no one was happy with the Might_Overrides name. So it was suggested that Overriding and Optional_Overriding replaces Overrides and Might_Override for the name of the pragmas.

Then the focus turned to the proposed wording. In particular, Erhard objected to the second sentence of this paragraph:
"A primitive operation to which pragma Overrides applies shall override another operation. This rule is not enforced at the compile time of a generic_declaration. In addition to the places where Legality Rules normally apply, this rule also applies in the private part of an instance of a generic unit."

There was much discussion on this point, which is best illustrated by this example:

```
generic
    type der is new T with …;
package Pac is
  type der2 is new der;
  proc p (A: der2);  -- 1
  proc p2 (A: der2); -- 2
  pragma Overriding (p2);
end Pac;

type BT is new T with …;
procedure p2 (A: BT); -- this is new method

package NewPac is new Pac (BT); -- 3
package NewPac1 is new Pac (T); -- 4
```

The presence of the pragma Overriding (in addition to Optional_Overriding) is a new language feature on generics that requires a primitive operation on the actual type in the generic instantiation. Hence, in the example above, declaration (2) is illegal for instantiation (4) because there is no primitive p2. To allow the instantiation (3) implies that the producer of the generic has some "prescience" that some subclasses will have primitive operations p2, whose semantics are such that the overriding makes sense. The latter needs to apply to all subclasses that might be defined in the far future, or else there is the risk of overriding semantically unrelated operations. The argument against accidental overriding speaks for making the overriding illegal, if T doesn't have the primitive operation p2 already. The argument for greater flexibility speaks for the rule as currently stated.

So the issue becomes the need for checks at both the generic definition and the instantiation. The declaration (2) above can be handled with a check at the generic declaration, while the declaration (1) can best be handled by checks at the instantiations.

So it was approved by 6-1-3 to drop the second sentence for this paragraph and the succeeding paragraph of the proposed additions to RM 8.3. The new wording for this paragraph are:

> "A primitive operation to which pragma Overriding applies shall override another operation. In addition to the places where Legality Rules normally apply, this rule also applies in the private part of an instance of a generic unit."
>
> "For a primitive subprogram inherited in a unit to which a pragma Explicit_Overriding applies, an overriding explicit declaration shall be a subprogram to which pragma Overriding or Optional_Overriding applies. In addition to the places where Legality Rules normally apply, this rule also applies in the private part of an instance of a generic unit."

Where does the Explicit_Overriding pragma apply? At the generic definition? At the instantiation of the generic? It makes most sense to make it apply to the definition for reasons that are analogous to the situation described above. The respective paragraph was rewritten several times. It was noted that 8.3(11) makes it clear that the rules must only apply to explicit subprogram declarations. The following version is presumed by the recorders to be the final version:

> "The configuration pragma Explicit_Overriding applies to all declarations within compilation units to which it applies, except that, in an instance of a generic unit, Explicit_Overriding applies if and only if it applies to the generic unit. At a place where a pragma Explicit_Overriding applies, an explicit subprogram declaration to which neither pragma Overriding or Optional_Overriding applies shall not be an overriding declaration. In addition to the places where Legality Rules normally apply, this rule also applies in the private part of an instance of a generic unit."

A couple of style changes were recommended:

- Replace use of "British spelling" with "alternative spelling".
- Replace the last sentence of the problem statement with

"All of these cases violate the Ada design principles of least surprise."

Approved the intent of the AI, 9-0-1. Randy will revise.

AI-222: Feature control

Erhard agreed with the general intent of the pragma, but objected to the language standard providing for a pragma that specifies the optional features the user intends to use, without also defining the feature identifiers in the standard. Otherwise, there is no portability gain and implementers might as well use an implementation-defined pragma. There is also the associated standard management problem when new features are added.

Tuck argued for the pragma, using the precedence of the Ada83 and Ada95 transition; vendors are/will be doing it (GNAT) and it nicely handles the optional Annexes.

An opposing view was that there are other existing mechanisms for vendors to let the user know the features the implementation does not provide, e.g., a syntax error (when a program tries to use the with_type_clause).

The opinions on this AI appeared to be evenly split as two straw polls showed on its inclusion in the standard (5-4-2) and its exclusion from the standard but with ARG recommendations to the vendors (5-4-2). Consequently, it was tabled by majority vote until more compelling events prove its worth.

AI-224: Unsuppress

Randy summarized the changes since the April 2000 meeting:

- Make the second parameter obsolescent because the semantics of this construct is poorly defined and apparently rarely used (UK reports that in 4000 instances of the Suppress pragma in 700K lines of code, there were only five uses of the second parameter). As an example of how poorly this feature is currently defined, Randy points out that it is not clear that the need to suppress the constraint check on the value of the expression to be assigned to an object can be done with the syntax ON => object. Furthermore, Randy stated that under these circumstances `Unsuppress (…, On => …)` is impossible to define.
- Remove the symmetry between Suppress and Unsuppress with respect to inheritance, namely, unlike Suppress, Unsuppress is not inherited in package body or subunit.
- Leave the meaning of Suppress and Unsuppress as configuration pragmas implementation-defined.

On point 1, it was recommended that there is no need to explicitly mention an obsolescence of the second parameter with the Unsuppress pragma in Annex J. Instead there should be an Implementation Permission that allows implementations to define and provide this feature. Also, a reference to the new alternative features in 11.5 should be made in the first paragraph of the addition to J.

Tuck wanted an additional wording to say that the checks that are (un)suppressed are not defined by the standard.

On point 2, it is very surprising that, without inheritance, in this example

```
package body P is
   pragma Suppress (all_checks);
   pragma Unsuppress (tag_checks);
   …
   procedure blah is separate;…
end P;
```

tag_checks are suppressed in blah depending on whether blah is "inlined" into the package or separate. One question is whether the pragmas' treatment of inheritance should be symmetric. The second question is whether inheritance should be permitted at all. Arguments for safety state that there be no inheritance but many implementations do already implement Suppress inheritance. Randy pointed out that implementations vary on how this is provided.

Tuck presented the case for limiting inheritance by saying that a configuration pragma overrides an inherited pragma and that the configuration pragma is overridden when an explicit pragma is specified. If a unit wants to ensure that it has the checks then the pragma Unsuppress must be explicitly specified and it won't be sabotaged by a configuration pragma. This appears to safely introduce Unsuppress with no changes to implementation regarding inheritance. While this appears to be a reasonable approach, it wasn't selling at the meeting, because the rules seemed overly complicated and because of the example above.

Tuck then tried a different approach, by starting with checking for possible agreement on any of the following points:

- Unsuppress revokes Suppress permission, including inheritance, namely, revocation overrides inheritance.
- Configuration pragmas override inheritance (of Suppress) pragma.
- A (Un)Suppress pragma inside a compilation unit revokes the opposite effect by inheritance or configuration pragmas.
- Drop inheritance requirement.

There seemed to be agreement on point 1 and 3, less agreement on point 4 but no convergence on point 2.

Erhard tried to sell the case for eliminating inheritance as the “safest” approach until a better one is proposed. Unfortunately, this approach is not upward compatible.

Tuck suggested four approaches to resolving this, as a way to get convergence:

- Require inheritance with weak configuration pragma (which is upward compatible)
- No inheritance, with configuration pragmas as starting point
- Inheritance is implementation-defined and configuration pragmas override inheritance (which is upward compatible)
- No configuration pragmas

Unfortunately this did not produce convergence either and the issue was tabled to let everyone sleep on it.

The next morning Tuck offered another view regarding the effect of inheritance and configuration with respect to these pragmas: Consider that Suppress pragma provides a set of permissions. A configuration Suppress provides an initial set of permissions to be applied to a compilation unit. The appearance of a Suppress pragma for a specific check in a compilation unit will add to the set of permissions for that compilation unit if the permission had not already been provided. The appearance of a Unsuppress pragma for a specific check in a compilation unit will revoke any permission to suppress that check for that compilation unit. It appeared that this insight was quietly received.

Randy will revise this AI accordingly.

AI-230: Implicit conversions between access types

Are there really two proposals in this AI, the name proposal (eliminating the need to declare access types for each type extension) and the conversion proposal? It is really meant to address conversion; the name benefit is just a by-product.

Erhard asked questions about the general use of this type beyond just conversion, such as in these situations:

- Passed as an actual value to a parameter of access T (of a primitive operation of T)
- Variables and parameters with assignments from specific access types to Q where Q is covered by T’Class (more general version of previous item)

What about conversion from T’Class_Access to specific access types? Tuck said this needs to be done explicitly.

Should implicit widening of access type conversion be applied across all designated types besides tagged types? There was actually an old Ada9X proposal to have T’Access allowed for all types for this purpose. It was dropped in the cut-down wars (with a small remnant remaining within records). Erhard opined that this is worth reconsidering to avoid patchwork amendments. But before discussing it, it was noted that the solution must address an ambiguity problem as illustrated by this example:

```
type t1 is …;
type t2 is new t1;

subtype at1 is access all t1’class;
subtype at2 is access all t2’class;

procedure p (x : at1);
procedure p (x : at2);

x: at2;

p (x); -- ambiguous, because of the opportunity of implicit type      --
conversion

y: t1’Class_Access;
```

```
p (y); -- ambiguous;
```

Tuck summarized that there are four problem areas that need to be addressed by this discussion:

- Too many explicit type conversions for non-controlling operands
- Calling wrong operations
- Too many named access types
- Impact of access types in a pure package

How would the proposal T'Access handle subtypes, in particular the constraints? The attribute applies to the first subtype. An example illustrates:

```
type A is array (1..10) of Integer;
type B is array (Positive range <>) of Integer;
subtype C is B (1..10);
```

`C'Access = B'Access`, this seems surprising given the constraint on B; but this seemed to be needed to support the generic contract model. And A'Access is not compatible with the others.

Further, this attribute causes problems when T is an access type, e.g., T'Access. Then T'Access'Access, etc., causes an infinite number of equality operators to be generated by some implementation approaches. T'Class_Access does not have this problem as access types are not tagged.

Tuck made a proposal (first with keyword "subtype", then with "type" and "all") that makes general access types implicitly convertible with other access types with the same designated types, in this fashion

```
type AccT is access all t;
```

AccT is implicitly convertible with other types that designate t. Widening the implicit conversion could be defined with

```
type AccT is access all T'Class;
```

This proposal is not upward compatible with Ada95 (but it is with Ada83) due to the overloaded procedure problem from the earlier example. Ouch!!! Some big test cases would help enlighten the ARG on the risks to the real world of this more general proposal.

What can be salvaged from all this discussion? Well, the AI does work for T'Class_Access and so the AI could be acceptable as is. Pascal Leroy will join Tuck is exploring the opportunity to expand this capability to all types.

AI-231: Access-to-constant parameters

There are two desires to be addressed: access to constants and access types that do not have a null value.

Pascal stated the proposed syntax of this AI is a curious way of specifying the need or assumption of non-null values. Mike Y. suggested that a subtype should be constrained to non-null values. This suggestion makes the existence of a simple access-to-constant with or without null values useful in its own right.

Some suggestions for declaring this non-null semantics:

```
    subtype AccNN is AccT not null;
```

And then this permits the following pseudonym:

`access T` is equivalent to `access all T non null`

A groan resulted when Erhard innocently asked for any impact to the T'Access proposal.

No constraints, like `not null`, should be applied to parameters of access types in a parameter_specification, following the rule of not applying explicit constraints on any parameter subtype. If constraints are needed, then do it in a subtype declaration.

The following syntax changes were proposed:

```
Access_definition ::= access all subtype_mark [not null] |
                      access constant subtype_mark [not null] |
                      access subtype_mark


Subtype_indication ::= … | subtype_mark not null
```

What happens to uninitialized variables of this non-null subtype? They get a default value of null and hence a constraint error is raised. This also presents compilers with an opportunity to give warnings/error messages, which also makes high integrity application developers happy. This also neatly handles instantiations of generics.

The AI should be tentatively renamed, "Improvements to Access Types".

This AI with the direction of providing a subtype with non-null constraint and the addition of an access constant parameter specification was approved 8-0-2. Tuck will rewrite this AI accordingly.

AI-232: Dispatching operation visibility and ambiguity

This was tabled, pending a separate AI from Tuck on his proposal for object'operator, which might solve the problem.

AI-234: Unsigned integer types

Voted "No Action", as this issue has been decided already, 9-0-1.

AI-241: Testing for Null_Occurrence

It became clear that the original AI was the wrong approach and the first message from Tuck (in the Appendix of the AI) about permitting the Null_Id value for Exception_Identity for Null_Occurrence instead of raising the exception should prevail.

This needs to be a Binding Interpretation. Para (14) should be split into two sentences.

Approved with these changes, 9-0-1. John will make the revision.

Detailed Review of Proposed Amendment AIs

Ravenscar Profile

Alan Burns presented the proposal for a Ravenscar Profile AI produced by the Tenth International Real-Time Ada Workshop. At the center of the proposal is the pragma Ravenscar which equates to a set of restrictions (see the proposal in the Appendix), with some of them being able to be overridden, namely,

- Task_Dispatching_Policy, which is defaulted to FIFO_Within_Priorities

- Locking_Policy, which is defaulted to Ceiling_Locking
- Task_Activation, being either Deferred or Standard

Discussion jumped right to the introduction of the third item, which certainly is not a restriction, but rather an extension to current language semantics. The goal of Task_Activation being Deferred is to make all other elaboration predictable without effects of task activation and to avoid certain elaboration errors. Here is an example of the need to hold off on the execution of the task body because a value is initialized by the body of package:

```
package P is
   type Device_Type is ...;
   task Device_Handler is
      pragma Priority( High );
   end Device_Handler;
end P;

with Device_Config;
pragma Elaborate( Device_Config );
package body P is
   D : Device_Type;
   task body Device_Handler is
   begin
      loop
         Process( D.Current_Data );
         Next_Frame := Next_Frame + Frame;
         delay until Next_Frame;
      end loop;
   end Device_Handler;
begin
   Device_Config.Initialize( D, ... );
end P;
```

The problem is that the Ada rules allow the task to begin to execute before the full program is elaborated. Hence, the task could access global data before it has been initialized. A similar problem arises for rendezvous attempted out of the activation code. Deferring the activation of the task until after the partition has been fully elaborated would avoid these problems. Unfortunately, this solution is not that straightforward. What happens if the sequence_of_statements of this package P contained an exception handler (rendering the changed behavior observable) or when the activation of the task object Device_Handler propagates an exception (again, with observable effects contrary to current activation rules)? These situations could be made bounded errors.

The discussion on this feature then turned towards whether it could exist outside of the Ravenscar pragma. The general opinion was that it could but that it implies a new scheduling policy. This policy would state that raising an unhandled exception or performing a rendezvous during elaboration is a bounded error. Unfortunately, the community served by the Ravenscar profile doesn't like the non-predictability of bounded error. Fortunately, bounded errors like this can be mapped onto a Program_Error, which works in this case and is required by Ravenscar, anyway.

The model is basically that there is an implied barrier synchronization of the task objects, just before activation, and for the environment task at the point where it invokes the main program (alternatively, when the main program reaches its "begin"). It seems acceptable to have such a notion of implicit synchronization points. The proposal for Deferred task activation also includes delaying the enabling of interrupts. Fortunately, this is semantically equivalent to the environment task performing its elaboration at the highest priority of the system; global task activation inherits this priority, too. So, these semantics could be part of the new scheduling policy.

Tuck disliked the Simple_Barrier_Variable, because simple expressions are needed there, not just a single variable. Replace the name with Simple_Barriers. Other restriction names needs wordsmithing, too, i.e., No_Requeues in place of No_Requeue.

Did the IRTAW consider excluding Streams or other sequential features that complicate finalization? The IRTAW dodged this issue with the No_Task_Termination restriction which avoids this problem. In general, the IRTAW focused on tasking and did not particularly examine sequential features, leaving that for individual implementations to consider.

The mention of No_Task_Termination restriction sparked a discussion of the goal of this restriction and how its semantics could be accommodated in the language standard. The main goal is to free the run-time system of direct and distributed overhead to implement task termination semantics. As a guaranteed compile-time enforced restriction it makes no sense, because this would require deciding the halting problem. Compile-time rejection under implementation-defined circumstances would be an awkward solution. In the absence of compile-time enforcement, a run-time semantics needs to be stated for the case when a task "runs out of code". There was a sentiment that, in this case, at least finalization should be guaranteed. The actual termination might then be considered a bounded error. However, the bounds are quite unclear. One possibility is that the task terminates silently. The presence of this restriction might imply that attributes such as 'Terminated are illegal or 'Callable always being true (which, of course, is incorrect, if the task terminated regardless). There are clearly a number of details that need to be ironed out before this capability can be codified into the Standard.

The ARG noted that more profiles could be anticipated, so maybe a more general pragma could be used, something like a pragma Execution_Profile or pragma Runtime_Profile or, preferably, simply Profile, whose argument identifies the specific profile. Those profiles with existing pragma can equate with this new pragma.

Joyce took the action item to revise the proposal in accordance with these discussions, pass it by the IRTAW participants for comment, and then produce the initial writeup of an AI.

Object'Operator

Tuck proposes an attribute notation for invoking a primitive operation of a type, where the parameter of interest is the first one, without having to specify the name of the package enclosing the type, nor even with-ing the package for the purposes of visibility. In other words, for the specification of a procedure in package P:

```
      procedure proc (x: t; y: u);
```

`a'proc (b)` is equivalent to `P.proc (a, b).`

There are problems with using attribute notation because it unnecessarily invades the name space of attributes. Vendors or the ARG could no longer add new attributes to the language without causing potentially massive incompatibilities with existing user code. The dot notation is the only viable choice among the two offered by Tuck. ":" was suggested as an alternative, but did not find many friends. Hence, now `a.proc(b)` is a pseudonym for `P.proc(a, b).` The dot also has the advantage (citing B. Meyer of Eiffel) that read-only data components and parameterless functions can be exchanged without affecting the code that uses the type. There are ambiguity issues regarding the names of primitive operations and components of the type but nothing more complicated than what exists already.

There appears to be no restriction on what the operand in this notation can be: tagged type, untagged type, class-wide type, function call. Overload resolution will require that the possibilities be limited to the primitive operations on the type of the prefix. Given that, the overload resolution of this notation is no more difficult than for the equivalent existing call notation. It also "happens" to nicely solve the problem that was addressed in Ada95 with the "use type" facility, as least as far as primitives that are not operators are concerned.

This appears to also solve the problem of AI-232. Erhard asked Tuck to write a new AI addressing tagged types first (so that this capability won't get lost in case the more general capability is not approved) and then attempt to apply it to untagged types as well.

Interfaces and Multiple inheritance

Tuck started the description of his proposal with the following example, which is first presented in its entirety and then repeated with explanations; the proposal is, to a very large extent, identical to the way interfaces work in Java.

```
type Int1 is [limited] interface;

procedure Notify (x: Access Int1; Y: TT) is abstract;

type Int2 is interface;

type NT is new T with Int1 and Int2 and record … end record;
procedure Notify (x: access NT; y: TT);

type Pint1 is access all Int1’Class;

I1 : Pint1 := new NT;

Notify (I1, z);
```

An “interface” is declared with this statement:

```
type Int1 is [limited] interface;
```

An interface does not declare data components. In many respects, this is very similar to declaring an abstract tagged null record. This implies that no interface objects may be created. The optional limited modifier has the same restrictions on interfaces that limited has on other types. Later, Tuck stated that non-limited tagged types are allowed to implement limited interfaces but not the reverse. If this is the case, the proposal still needs to explore possible inconsistencies, especially in polymorphic settings.

Following the interface declaration would be the declaration of primitive operations of the interface:

```
procedure Notify (x: Access Int1; Y: TT) is abstract;
```

Initially, Tuck thought to permit concrete implementations as well as abstract operations. But subsequent discussion on this point concluded that there was no compelling reason to permit this, especially in light of the implementation complexities and the obvious limitations due to the absence of data components. Hence all primitive operations for an interface must be specified as abstract.

The next pair of declarations describe how to implement a mix-in of interfaces with a tagged type:

```
type NT is new T with Int1 and Int2 and record … end record;
procedure Notify (x: access NT; y: TT);
```

The type NT is specified to implement both interfaces, Int1 and Int2. It must then supply primitive operations (either new operations or inherited/overridden operations from T) for all operations of Int1 and Int2 with matching profiles (with NT substituted for either Int1 or Int2) ; presumably explicitly repeating matching abstract operations would also suffice, with the consequence that NT is abstract. In all other respects, NT looks like any other tagged type.

To facilitate the use of an interface, like Int1, an access type to the class of Int1 interface is declared:

```
type Pint1 is access all Int1’Class;
```

This is similar to the use of classwide access types. In particular, objects of such access type can be declared and dispatching calls on operations of Int1 issued. An open issue is whether operations with parameters of class-wide interfaces can be specified; presumably the answer is “yes”.

The following declaration provides for the creation of an object of interface-wide access type and the assignment of the reference to a dynamically allocated object of type NT:

```
I1 : Pint1 := new NT;
```

Until I1 is assigned a different value, all primitive operations of interface Int1 invoke the associated operation of NT, as the last line of the example shows:

```
Notify (I1, z);
```

Here the primitive operation of Int1, Notify, indirectly invokes the Notify operation of NT because that is the NT operation onto which the interface operation is mapped.

Some options on this example were discussed. One option would be that NT is a root type, such as,

```
type NT is tagged with Int1 and Int2 and record … end record;
```

Another option would be to permit an interface to provide one or more other interfaces, such as,

```
type I is [limited] interface with IntA and IntB;
```

Here the declaration means that I covariantly "inherits" all the operations of interfaces IntA and IntB. This permission has impact on the dispatch tables for I (see below).

Discussion on the reserved word **`interface`** found some problems with it. It collides with the obsolescent pragma Interface and it represents a new reserved word. Pascal suggests **`abstract`** instead, such as:

```
type Int1 is abstract;
```

Presumably, an AI for this proposal will resolve this issue.

Tuck then explained how this would be implemented. Every interface-wide object would consist of a reference to the implementation object and a reference to a dispatch table associated with the implementation of the interface by the concrete tagged type. The example declaration:

```
I1 : Pint1 := new NT;
```

illustrates the details of the interface-wide object. The object that is referenced is an NT object, where the NT object contains the "tag" associated with NT and the value of the object. The NT object is like any other tagged object; the "tag" of course is a reference to the dispatch table associated with NT. The dispatch table for NT must contain references to operations that match the operations of interface Int1 (as well as the matching operations for Int2). The "implementation dispatch" table for Int1 then is responsible for mapping the operations of Int1 to the matching entries in the dispatch vector of NT. This type of "fat" pointer is similar to how implementations might represent a constrained access to an unconstrained array object. The layout of the dispatch table of NT first maps all the operations of the interface, Int1, onto matching concrete operation of NT then does likewise for the operations of interface, Int2. The implementation dispatch tables for Int1 and Int2 are created by copying references to the matching operations of NT when the type NT is declared; copying is necessary because the ordering of the NT dispatch table can't be guaranteed to match the ordering of the Interface implementation dispatch tables.

The layout of the implementation dispatch table is determined by operations of the interface that are unique to that interface and the operations that the interface "inherits" from the other interfaces being implemented. Apparently, if the interface Int1 was declared as follows:

```
type Int1 is [limited] interface with IntA and IntB;
procedure Notify (x: Access Int1; Y: TT) is abstract;
```

then a possible layout of the implementation dispatch table for Int1 would place the operation of IntA first in the table followed by the operations of IntB and then finally the Notify operation. Extrapolating this to a an arbitrarily deep graph of interfaces, the layout would possibly be laid out as a flattened graph.

Clearly, there are lots of questions to be answered:

- How are type conversions and memberships handled among interfaces types? It may be easy to see how conversion can be done up the interface tree; it is not clear how conversions can be done down the interface tree.

Tuck believes semantically that the concept of “covering” with respect to tagged types can be adapted straightforwardly to describe the conversion and membership semantics
- How will interfaces be handled by generics? Will it require a new formal mechanism or would it be possible to reuse tagged type mechanisms?

Tuck is asked to proceed on writing an AI.

Standard Library Operations

Randy has proposed a set of operations that are not like any existing standard because the ones he examined, like POSIX, have problems. Mike Yoder provided further evidence that POSIX, despite having an Ada binding, would be a significant mistake to use as a basis/replacement of Randy’s proposal.

Tuck suggested that Randy expand his investigation to look as LDAP and JNDI (Java Naming and Directory Interface) as another guideline for modeling.

Randy is tasked to make this into an AI (AI-248).

Object_size Attribute

Writing up a proposal on this subject is on Randy’s to-do list.

Detailed Review of Non-Amendment AIs

AI-195: Streams

The heart of the change is that you get stream elements of 1 byte, 2 bytes, 4 bytes, 8 bytes, 12 bytes, etc.; there are no 3 byte elements! The intent is correct but the wording needs work.

Some consistency between use of bits and T’Stream_Size is needed. Bits appears to be the rule for specifying the requirement because users typically view this level of capability in bits.

The AI should mention that ‘Stream_Size is an operational attribute (note the inheritance rules!).

It was suggested that there might be a need for an upper bound on a specifiable ‘Stream_Size, e.g., largest supported size for the respective kind of type. This was discussed controversially.

There are objections to the use of “rounding up”. Many feel that its use is too pedestrian for use. On the other hand, there is precedence in the RM (13.2 (9)) for using this term; hence it remains.

There is some discussion over the parenthetical phrase in the last new paragraph, which addresses how to handle values that are outside the range of a first subtype. For the constraint `range 0 .. 127,` what happens to the extra bit in an 8 bit representation? Sign extend? Zero fill? With very specific wording, Tuck aims to add details to make the portability of streams possible; this is something that really needs to be addressed very soon. Why not an outside canonical form for the mechanism for this portability? Well, for one thing, there are competing forms from Sun and HP. Open question??

The intent of the discussion is approved and Tuck is asked to wordsmith this AI.

AI-227: Behavior of Ada.Streams.Read when at the end of stream

Dropping the answer in the answer for the question section actually improves the description of the problem.

Approved with editorial change, 6-0-1. Randy will revise accordingly.

AI-235: Resolving 'Access

Pascal claimed that changes to 3.10.2 (27/1) now prevent the form X'Access from being used for dispatching, namely for the type T'Class. This surely was not intended.

No one could parse the recommended wording:

> "For an attribute_reference with attribute_designator Access (or Unchecked_Access -- see 13.10), the expected type shall be a single access type whose designated type covers the type of the prefix, or, if the type of the prefix is D'Class, whose designated type is D, or whose designated profile is type conformant with that of the prefix. [The prefix of such an attribute_reference is never interpreted as an implicit_dereference or parameterless function_call (see 4.1.4).] The designated type or profile of the expected type of the attribute_reference is the expected type or profile for the prefix."

A previous version appeared to be better written:

> For an attribute_reference with attribute_designator Access (or Unchecked_Access -- see 13.10), the expected type shall be a single access type whose designated type covers the type of the prefix, or whose designated profile is type conformant with that of the prefix. [The prefix of such an attribute_reference is never interpreted as an implicit_dereference or parameterless function_call (see 4.1.4).] The designated type or profile of the expected type of the attribute_reference is the expected type or profile for the prefix."

The use of bullets helps improve the readability of this paragraph.

> "For an attribute_reference with attribute_designator Access (or Unchecked_Access -- see 13.10), the expected type shall be a single access type A such that:
> - A is an access-to-object type with designated type D and the type of the prefix is D'Class or is covered by D; or
> - A is an access-to-subprogram type whose designated profile is type conformant with that of the prefix.
>
> [The prefix of such an attribute_reference is never interpreted as an implicit_dereference or parameterless function_call (see 4.1.4).] The designated type or profile of the expected type of the attribute_reference is the expected type or profile for the prefix."

The addition of the phrase "the type of the prefix is D'Class" is designed to solve the dispatching problem.

Approved with changes, 8-0-2. Randy will change.

AI-236: Minimum criteria for metrics documentation

Approved without discussion, 8-1-0. Randy will run a spellchecker on it.

AI-237: Finalization of task attributes

Mike verbalized three points on this AI:
- there still needs to be a hook to clean up the user-defined attributes to avoid memory leaks
- there needs to be a solution to the apparent race condition between the master of the attribute instantiation and the task with the attribute.
- Current_task is undefined, making its use a bounded-error during finalization of the attributes

The first point seems to be covered already but the second one appears to need some guidance, such as limiting the action to only one termination. The other two points were addressed as follows:

Add the following after C.7.2(29):

> "After a task terminates, an implementation may finalize all attributes of the task instead of when the master of the instantiation is finalized, and reclaim any other storage associated with the attributes."

And then after C.7.1 (17) add a new case

> "finalization of user-defined attribute (see C.7.2)"

The intent of these changes was approved with subsequent editorial review, 9-0-0. Randy will revise accordingly.

Appendix: Ravenscar Profile Proposal

!topic Addition of support for Ravenscar Profile
!reference RM-D
!from Brian Dobbing 00-09-24, Alan Burns 00-11-17, Joyce Tokar 00-11-17
!keywords Ravenscar, high integrity
!discussion
This comment proposes the addition of support in the language standard for the *de facto* standard Ravenscar Profile. This profile is included in the ISO document "Guide for the use of the Ada programming language in high integrity systems".
The comment proposes addition of pragma Ravenscar to Annex D of the RM as follows.
Pragma Ravenscar defines an alternative mode of operation that consists of the following amendments to the dynamic semantics of the standard.
[Note: I say "alternative" rather than "non-standard" as in section 1.5. This is because its inclusion in the standard makes it not non-standard.]

Static Semantics
pragma Ravenscar;
Pragma Ravenscar is a configuration pragma that applies to an active partition.

Dynamic Semantics
1 Task Dispatching Policy
The default Task_Dispatching_Policy for the active partition is FIFO_Within_Priorities. An implementation may define alternative task dispatching policies.

2 Locking Policy
The default Locking_Policy for the active partition is Ceiling_Locking. An implementation may define alternative locking policies.

3 Restrictions pragmas
3.1 Standard Pragmas
The following pragma Restrictions identifiers defined in the standard apply to the alternative mode of operation defined by pragma Ravenscar:
Max_Asynchronous_Select_Nesting => 0
Max_Task_Entries => 0
Max_Protected_Entries => 1
No_Abort_Statements
No_Asynchronous_Control
No_Dynamic_Priorities
No_Implicit_Heap_Allocations
No_Task_Attributes
No_Task_Allocators
No_Task_Hierarchy
The pragma No_Task_Hierarchy imposes the constraint that all tasks must depend *immediately* on the Environment task as a result of all task objects being created by library level declarations.
[Note: It is not clear to me whether the existing RM wording fully implies this. It is not sufficient to say that all tasks shall depend (directly or indirectly) on the Environment task since this presumably would include those declared inside library-level subprograms. What we want to restrict is having to support "masters" and "waiting for dependent tasks"].
3.2 Additional Pragmas
The following new pragma Restrictions identifiers are defined as applying to the alternative mode of operation defined by pragma Ravenscar:
[Aside: Alan Burns has a more complete write-up of these pragmas than is presented here.]
Max_Entry_Queue_Depth = 1
Max_Entry_Queue_Depth defines the maximum number of calls that are queued on a (protected) entry. Violation of this restriction results in the raising of Program_Error exception at the point of the call.

[Note: These semantics are intended to be consistent with those for blocking on a suspension object via Suspend_Until_True.]

No_Calendar

There are no semantic dependencies on package Ada.Calendar.

No_Dynamic_Attachment

There is no call to any of the operations defined in package Ada.Interrupts (Is_Reserved, Is_Attached, Current_Handler, Attach_Handler, Exchange_Handler, Detach_Handler, Reference)
[Note: Static attachment via pragma Attach_Handler is supported.]
{Note: At IRTAW-10, we called this No_Dynamic_Interrupt_Handlers, but that doesn't really convey the correct meaning of the restriction, hence my alternative suggestion.]

No_Local_Protected_Objects

All protected objects are created via library-level declarations.
[Note: This matches the additional semantics for No_Task_Hierarchy. If these additional semantics were considered to be unacceptable, we could introduce No_Local_Task_Objects to go with this one.]

No_Protected_Type_Allocators

There are no allocators for protected types or types containing protected type components.
[Note: This is like No_Task_Allocators]

No_Relative_Delay

Delay_relative statements are not allowed.
[Note: Unfortunately, No_Delay in H.4 is too strong since we want to allow delay_until Ada.Real_Time.Time, but not relative delay (non-deterministic) or Ada.Calendar (too coarse).]

No_Requeue

Requeue statements are not allowed.

No_Select_Statements

Select_statements are not allowed.
[Note: This includes selective_accept (there are no task entries anyway), timed and conditional entry calls, and asynchronous_select (which is re-inforced by Max_Async_Select_Nesting = 0.]

No_Task_Termination

No_Task_Termination defines task termination (including for the Environment task) to be a bounded error condition. The possible effects are:

> The task terminates as defined by the standard mode of operation.
> Prior to termination as defined by the standard mode of operation, the task calls an implementation-defined subprogram. The mechanism for specifying the subprogram to be called, and any restrictions on the operation of the subprogram, are implementation-defined.

[Note: This is defined so as to mitigate the hazard of "silent task death" in high integrity systems and to provide the opportunity for application-specific recovery action.]

Simple_Barrier_Variables

The Boolean expression in an entry barrier shall be the value of a Boolean component of the enclosing protected object.

4 Outstanding Issue -- Task_Activation

To satisfy the requirements of the Safety Critical and High-Integrity domains, there is a need to define the behavior of program elaboration to be atomic. That is to say, that no interrupts are delivered during this period of execution. In addition, task activation shall be deferred until the completion of all elaboration code.

A proposed approach to addressing this concern is to introduce an optional argument on the pragma Ravenscar such that the activation of library level tasks may be specified. This would have the effect of the modifing the syntax and semantics for pragma Ravenscar as follows:

Static Semantics

pragma Ravenscar[(Task_Activation => Deferred | Standard)];

Dynamic Semantics

4.1 Task_Activation => Deferred

With the Deferred value for the Task_Activation argument, all task activation and all interrupt handler attachment for library-level tasks and interrupt handlers are deferred. The deferred task activation and handler attachment occurs

immediately after the “begin” of the Environment task. At this point, the Environment task is suspended until all deferred task activation and handler attachment is complete.
In this mode of operation, it is a bounded error for the Environment task to execute a call to Ada.Synchronous_Task_Control.Suspend_Until_True or to execute a protected entry call that is blocking during its declarative part. Program_Error may be raised by the call, or the active partition may deadlock.
[Note: In either case, the active partition is terminated.]
It is a bounded error for any deferred task activation to fail. The Environment task and all tasks whose activations fail are terminated. A task whose activation succeeds may continue to execute, or may instead be immediately terminated, thereby completing execution of the partition.
[Note: The preference is to terminate the partition immediately to mitigate the hazard posed by continuing execution with a subset of the tasks being active. However this would introduce code into the runtime to detect this corner case, so it is not mandated.]

4.2 Task_Activation => Standard

This value defines the execution of the declarative part of the Environment task to be as defined by the standard mode of operation with respect to task activation and interrupt handler attachment.