Minutes of the 45th ARG Meeting

11-13 November 2011

Denver, Colorado, USA

Attendees: Steve Baird, John Barnes, Randy Brukardt, Gary Dismukes, Brad Moore, Erhard Ploedereder (except Sunday), Jean-Pierre Rosen, Ed Schonberg, Tucker Taft.

Observers: Greg Gicca (Friday only).

Meeting Summary

The meeting convened on 11 November 2011 at 09:15 hours and adjourned at 12:00 hours on 13 November 2011. [Too bad we didn't start at 11:11 hours. - Editor.] The meeting was held in the Glenarm room of the Magnolia Hotel in downtown Denver, Colorado. The meeting covered the entire agenda of Ada 2005 AIs, most of the Ada 2012 AIs, along with a variety of other issues that had been previously brought up only in e-mail.

AI Summary

The following AIs were approved:

AI05-0264-1/02 Add missing commas and other editorial changes in standard (9-0-0)

The following AIs were approved with editorial changes:

AI05-0262-1/04 Rewordings from the Second Editorial Review (9-0-0)
AI05-0265-1/01 The location of tampering checks should be specified (8-0-1)
AI05-0266-1/01 Use the latest version of ISO/IEC 10646 (7-0-2)
AI05-0267-1/01 Improvements for aspect specification (8-0-1)
AI05-0268-1/01 Examples in new sections of the Standard (8-0-0)
AI05-0272-1/01 Pragma and Attribute restrictions (7-0-0)
AI05-0273-1/01 Problems with the Old attribute (9-0-0)
AI05-0274-1/03 Side effects during assertion evaluation (6-1-2)
AI05-0276-1/01 Violation of predicate legality rules in generic units (8-0-1)
AI05-0277-1/01 Aliased views of extended return objects (8-0-1)
AI05-0278-1/01 Set_CPU called during a protected action (6-0-3)
AI05-0279-1/01 Renaming an invalid Unchecked_Conversion result (7-0-0)

The following AI was discussed without a conclusion:

AI05-0284-1/00 Accessibility of anonymous access returns

The following AIs were approved with editorial changes, including converting them to Ada 2005 AIs:

AI12-0006-1/01 Accessibility of null (7-0-2) [now AI05-0270-1]
AI12-0007-1/01 Accessibility of access discriminants of a subtype (5-0-4) [now AI05-0281-1]
AI12-0008-1/01 Bad ancestor constraints (6-0-2) [now AI05-0282-1]
AI12-0011-1/01 Behavior of Random.Reset (9-0-0) [now AI05-0280-1]
AI12-0012-1/01 Failure behavior of Directories.Create_Path (7-0-2) [now AI05-0271-1]
AI12-0013-1/01 More issues with the definition of volatile (7-0-2) [now AI05-0275-1]

The following AI was discussed and converted to an Ada 2005 AI:

AI12-0010-1/01 Stream_IO should be preelaborated [now AI05-0283-1]

The following AIs were discussed and a resolution was postponed until after the Ada 2012 revision is completed:

AI12-0009-1/01 Iterators for Directories and Environment_Variables
AI12-0015-1/00 Ada unit information
AI12-0016-1/00 Implementation model of dynamic accessibility checking

Detailed Minutes

Welcome

Joyce Tokar (Convenor of WG 9) opens the meeting with some brief remarks. She thanks us for our hard work on the Ada 2012 standard, and describes the current standardization plan. We are going to do another ARG review of the draft Standard concurrently with the National Body review, with the intent of submitting it using an experimental standardization process in the spring. That process should get us to a completed Standard very quickly. The review deadline will be January 15th.

Previous Meeting Minutes

John says he didn't have any changes other than one he doesn't understand. We vote, then John announces that he has figured out his note: the title of AI05-0253-1 is wrong.

Approve minutes with change: 9-0-0.

Date and Venue of the Next Meeting

We should have a February meeting after all, to dispose of National Body and ARG comments on draft 14, before submitting it to a WG 9 vote for approval. We discuss a variety of possible locations. We'd like warm (not New York – cold & expensive), and reasonable international access. We don't want to impose on Greg again (but that would make a reasonable fall-back position). Joyce left quickly, maybe that's why. (Phoenix would make sense.) Ed will contact some AdaCore people to see if they could host.

ASIS

WG 9 has instructed us not to spend any more effort on an ASIS standard until there is more user demand for that standard. So we are not going to require any further ASIS work of ARG members (they are welcome to do some if they like); Randy will file the work that's already been done so it is not lost.

Ada 2012 Review

We're going to do another level of review, with a twist: ARG members will be required to produce at least 2 ACATS tests (1 B, 1 C) for some clause that they are reviewing. The idea is that writing examples is valuable, and if that's done, they might as well be executable and in ACATS form. Randy will stand ready to assist anyone with the requirements of ACATS tests. Reviews and tests will be due January 15th.

ARG Membership

Ed would like to add Geert Bosch to the ARG. He explains that Geert is an expert on machine architecture and processor-level synchronization issues. There are no objections from the group.

Ed asks Randy to add Geert to the ARG mail list as soon as possible.

Thanks

Thanks to Ed Schonberg for chairing the meeting.

Thanks to SIGAda for hosting our meeting.

John Barnes thanks the kitchen for his fried eggs, and thanks the group for helping on the Rationale. [Yes, he really said that first part - Editor.]

Thanks to Randy Brukardt for taking minutes and editing the standard. [And running the last hour of the meeting after Ed left to catch his flight.]

Old Action Items

With the hiatus of the ASIS work, there are no known open action items.

New Action Items

The combined unfinished old action items and new action items from the meeting are shown below.

Everyone:

- Review Draft 14 of the Standard as assigned. Due January 15th, 2012.

Randy Brukardt:

- Apply the changes made at this meeting to the draft Standard, along with editorial comments from the ARG and public.

Editorial changes only:

- AI05-0262-1
- AI05-0265-1
- AI05-0266-1
- AI05-0267-1
- AI05-0268-1
- AI05-0272-1
- AI05-0273-1
- AI05-0274-1
- AI05-0276-1
- AI05-0277-1
- AI05-0278-1
- AI05-0279-1
- AI05-0284-1
- AI12-0006-1 [now AI05-0270-1]
- AI12-0007-1 [now AI05-0281-1]
- AI12-0008-1 [now AI05-0282-1]
- AI12-0010-1 [now AI05-0283-1]
- AI12-0011-1 [now AI05-0280-1]
- AI12-0012-1 [now AI05-0271-1]
- AI12-0013-1 [now AI05-0275-1]

Detailed Review

The minutes for the detailed review of AIs and SIs are divided into ASIS Issues (SIs), Ada 2005 AIs, and Ada 2012 AIs (no SIs were considered at this meeting). The AIs and SIs are presented in numeric order, which is not necessarily the order in which they were discussed. Votes are recorded as "for"-"against"-"abstentions". For instance, a vote of 6-1-2 would have had six votes for, one vote against, and two abstentions.

If a paragraph number is identified as coming from the Ada 2012 Standard, the number refers to the text in Draft 14 of the Ada 2012 AARM. Paragraph numbers in earlier drafts may vary.

Detailed Review of Ada 2005 AIs

AI05-0005-1/00 Editorial comments on AARM 2005

[Editor's note: This is an after-the-fact assignment of this topic to an AI.]

Steve had sent mail which says that 10.2.1(18.a/2) is misleading, because it does not consider the possible effects of indirect calls. It is possible using dispatching or access-to-subprogram calls to call an impure subprogram from inside of a pure function.

Randy notes that this is an important and widely-used technique. For instance, he often passes an access to a logging routine to preelaborated and pure subprograms in order to provide debugging output. (I/O is neither pure nor preelaborated – but see the discussion of AI12-0010-1.) So we do not want to try to close the "hole".

We agree with Steve that the AARM note ought to reflect this technique and its consequences. We'll apply Steve's suggestions to the AARM note.

This will be added to AI05-0005-1 (the AARM note AI); there won't be a separate AI for it.

Approve 9-0-0.

AI05-0262-1/04 Rewordings from the Second Editorial Review

The paragraph numbers in this AI are those from Draft 13 (the one reviewed by the second editorial review). The minutes only record wording that we actually decided to change; we reviewed all of the other wording changes that involved semantic changes (these are all listed in the !discussion section) and decided to leave them unchanged.

6.1.1(29/3) [31/3 in Draft 14]. Replace the wording with:

> If a precondition or postcondition check fails, the exception is raised at the point of the call; Redundant[the exception cannot be handled inside the called subprogram or entry.] Similarly, any exception raised by the evaluation of a precondition or postcondition expression is raised at the point of call.

6.1.1(33/3) [35/3 in Draft 14].

This should be “unspecified”, rather than “implementation defined”, as this would be hard to describe and not something that anyone should depend on anyway.

10.1.1(12.2/3), (12.3/3): AI-262 should be in the reference list (in the AARM) for these paragraphs. Modify the text as follows:

- For each [nested] package_declaration {occurring} immediately within the visible part, a ...
- For each type_declaration {occurring} immediately within the...

A.4.11(49/3): Tucker suggests: “a UTF_String, UTF_8_String, or UTF_16_String”. Similarly in 50/3, 51/3.

Randy asks if “code position” should be changed elsewhere in the standard to “code point” (which is the 10646 term). 3.5.2, 3.5, 2.1, 1.1.4. Tucker says that this is about 5 places. Straw poll that we should change: 5-0-4.

A.18.2(88/3): We try to improve the wording by dropping “exactly” and replacing “of the vector” with “of Vector”.

Brad says that this wording does not allow changing the length of the Vector. We want the length to be changeable (that's what the GNAT implementation does, and it's surely more flexible that way), so the length should be written. Gary and Steve also note that Vector is a type, not an object, so most of this wording doesn't make any sense.

We'll take this wording off line, and Brad gets the short stick in the eye, ummm, the action item to create wording. Tucker suggests checking the other containers as well, since Trees don't have a Length.

In the discussion, A.18.2(252/3) should be 254.

Ed had asked about the meaning of reverse iteration. He says that he withdraws that question, as Matt's response convinced him that we have it right.

Ed asks if we should add an iterate with a Start to the ordered map and set. (Matt asked this in e-mail yesterday). This seems like a good idea; add that to this AI.

Randy notes that Matt had another question. What should Start => No_Element do? Matt suggests that No_Element do nothing. He thinks that it should OK to pass Next(X) (which might return No_Element). Tucker says that for such an example to make sense, there has to be some sort of outer loop that is using Next(X). But in that case, the concern doesn't make sense, because you have to make a check somewhere to exit the outer loop, and that check would prevent passing No_Element to the inner loop. So he thinks that raising Constraint_Error is the best result.

The standard currently says that it iterates the whole thing, we'll change that to propagates Constraint_Error because there several possibilities (iterate everything, iterate nothing, and raises an exception) and we can't decide which makes the most sense. There are always examples that a particular answer works well with. But in this case, starting nowhere doesn't (usually) make sense – the parameter name is "Start", not "Someplace_in_this_Container_or_not".

B.2(10.1/3) should be at the start of the annex, after B(1). Perhaps we need an AARM note to say that this should not be taken to mean that you can avoid implementing something that is required in this annex. (It is not an optional annex like the specialized needs annexes.) John notes that there already is a similar note here. Probably we just need to say this applies only to the optional features by adding "optional" in front of the first use of "aspect".

Looking at C.7.1(14). We don't want to re-evaluate the barrier for each caller, which would be necessary if E'Caller was allowed in the barrier. No compiler tested allowed the attribute in an entry barrier. (Note to editor: this should be documented as an incompatibility, but not expected in practice – because compilers already reject it).

Erhard says that he does not like this wording, as this isn't a parenthetical remark. Use instead:

"...inside an accept_statement, or entry_body after the entry_barrier, corresponding..."

The comma usage is weird. We all turn to Gary, Mr. Comma. Gary says it is OK (he doesn't love it).

We'll defer a vote on this AI until we have wording from Brad for the streaming of containers.

On Saturday, we continue the discussion of AI05-0262-1:

John noted that 3.2.4(12/3) says “both” operands, which seems to not allow “**not**”. The intent is that all of the logical operators are allowed.

Replace “both” with “all”, or “each” or “every”. “...each operand is...”

This allows “**not**”.

Brad says that the AARM ZIP file is missing the html page for 3.2.4. Randy will investigate.

Steve wonders why if_expressions are not included in predicate static; it's odd that only case_expressions are allowed.

This was left out on purpose because it gets complicated. We wanted the result to be representable in a small set of ranges. It's the same reason that we only allow the current instance in a case expression. The conversion of a case expression to an equivalent if expression would only allow a boolean expression, which is just about pointless.

Why is case allowed? Probably because of the completeness check for enumerations. Bob would know, but he's not here.

Turning to the streaming for containers. Brad sent the following message about rewordings for that text:

I have two possible wordings to look at. We could also create a hybrid taking the best of both worlds if necessary.

The first version is longer but more accurately describes what's going on. The second is shortened, and leaves more to the imagination.

I'm not sure if we want to talk about writing the Length to the stream. I could imagine some implementation that might not actually write the length to the stream, but instead uses some sort of "end of elements" marker. I think this would be a poor implementation though, as it would be more efficient to effectively call <V>.Insert_Space(Count => length) upon determining the length from the stream for 'Read.

I have seen "Execution of the Write attribute for a" wording elsewhere in the RM so I am wondering if we need to be that formal. I also was influenced by the wording for 'Input and 'Output in the Streaming section.

A.18.2(88/2)

Execution of the Write attribute for a <Vector> object <V> writes the value of <V>.Length() to the stream, and writes <V>.Length() elements of the vector to the stream. It may write additional information about the <vector> as well. Execution of the Read attribute for the <Vector> subtype reads and returns a <Vector> object from the stream, using the value of Length() written by a corresponding <Vector>'Write to determine the number of <vector> elements to read from the stream. Any additional information written by <Vector>'Write is also consumed by <Vector>'Read.

or

<Vector>'Write for a <Vector> object <V> writes <V>.Length() elements of the vector to the stream. It may write additional information about the <vector> as well. <Vector>'Read reads and returns a <Vector> object from the stream, and consumes any additional information written by <Vector>'Write.

Read is a procedure, so it doesn't return anything. We word-smith this for quite a while, and come up with:

<Vector>'Write for a <Vector> object <V> writes Length(<V>) elements of the <vector> to the stream. It also may write additional information about the <vector>.

<Vector>'Read reads the representation of a <vector> from the stream, and assigns to *Item* a <vector> with the same length and elements as was written by <Vector>'Write.

Approve AI with changes: 9-0-0.

AI05-0264-1/02 Add missing commas and other editorial changes in standard

Approve AI: 9-0-0.

AI05-0265-1/01 The location of tampering checks should be specified

Tucker suggests "{When}[If] tampering with cursors is *prohibited* for a particular <container-kind> object <M>. ...". This needs to be done in both places in that text.

Steve would like some statement about having no other effect.

So add after the Program_Error statement:

"... Program_Error is propagated by {a call of} any language-defined subprogram that is defined to tamper with the cursors of <M>{, leaving <M> unmodified}."

In both places, of course.

Tucker suggests adding redundant text to the last sentence (or tamper with the cursors of V). Randy notes that we had that elsewhere originally.

That makes the entire text:

> When tampering with cursors is *prohibited* for a particular <container-kind> object <M>, Program_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the cursors of <M>. Similarly, when tampering with elements is *prohibited* for a particular <container-kind> object <M>, Program_Error is propagated by a call of any language-defined subprogram that is defined to tamper with the elements of <M> (or tamper with the cursors of <M>), leaving <M> unmodified.

Turning to all of the Process.**all** wording. We ought to mention that the prohibition is during a call to Process.**all**, otherwise there could be confusion with the evaluation of the access object.

"...is prohibited during the execution of {a call on} Process.**all**."

Use “the call on” for those routines with a single call, like Query_Element and Update_Element.

Tucker suggests using “invocation” instead:

"...is prohibited during the {invocation}[execution] of Process.all."

Randy notes that this term ("invocation") is not defined by the standard. He thinks it is better to use “of a call on”. The group agrees.

In the wording change for A.18.19(10/3) and others, Erhard notes that “in the middle of an operation” is not needed anymore.

> It is a bounded error to use a bounded vector if it was the target of an assignment_statement while tampering was prohibited for elements [or cursors] for the source of the assignment. Either Program_Error is raised, or the operation proceeds as defined.

Tucker says that “source” is unfortunately not defined. We have to say “source expression”.

Tucker notes that this should be bounded such that when the target is re-assigned, this no longer is the case.

He also worries that there are other assignments to which this should apply (like an aggregate component).

He suggests:

> It is a bounded error to use a bounded vector if its value was the result of an assignment from an object for which tampering was prohibited with elements [or cursors]. Either Program_Error is raised, or the operation proceeds as defined.

Steve worries that this needs to say at the point of the assignment, not the current state. He suggests:

> It is a bounded error to use a bounded vector if its value was the result of an assignment from an object for which tampering with elements [or cursors] is prohibited at the time of assignment. Either Program_Error is raised, or the operation proceeds as defined.

Tucker suggests:

> It is a bounded error to use a bounded vector if its value was the result of an assignment from an object while tampering with elements [or cursors] of that object is prohibited. Either Program_Error is raised, or the operation proceeds as defined.

Steve doesn't think this works. We break for lunch, which has already arrived in the meeting room.

Lunch discussion of this AI proves we need to assign some work. Tucker, Randy, and Steve will work this out.

On Saturday, after the three Amigos spent more than an hour of our extended lunch discussing this, the following wording changes were suggested:

Add an AARM note after A.18.2(93.1/3):

AARM Ramification: We don't need to explicitly mention assignment_statement, because that finalizes the target object as part of the operation, and finalization of an object is already defined as tampering with cursors.

[Make a similar change in the tampering with cursors rules for all other containers.] This question comes up periodically, since it appears to be missing, so we add a note to ensure that we can read the answer the next time the question is asked.

Add in A.18.19: (Replacing the existing bounded error.)

Bounded (Run-Time) Errors

> It is a bounded error to assign from an object while tampering with elements Redundant[or cursors] of that object is prohibited. Either Program_Error is raised by the assignment, the target object prohibits tampering with elements Redundant[or cursors], or execution proceeds normally.

Erroneous Execution

> When a bounded vector object V is finalized, if tampering with cursors is prohibited for V other than due to an assignment from another vector, then execution is erroneous.
>
> AARM Reason: This is a tampering event, but since the implementation is not allowed to use Ada.Finalization, it is not possible in a pure Ada implementation to detect this error. (There is no Finalize routine that will be called that could make the check.) Since the check probably cannot be made, the bad effects that could occur (such as an iterator going into an infinite loop or accessing a non-existent element) cannot be prevented and we have to allow anything. We do allow re-assigning an object that only prohibits tampering because it was copied from another object as that cannot cause any negative effects.

[Add similar text to every bounded container.]

Tucker suggests and improvement to the bounded error text:

> Either Program_Error is raised by the assignment, execution proceeds with the target object prohibiting tampering with elements Redundant[or cursors], or execution proceeds normally.

Brad wonders if "when" is temporally wrong for the erroneous execution text; the erroneousness appears only to apply during the finalization. Tucker explains that once execution is erroneous, it stays that way.

Approve AI with changes: 8-0-1.

AI05-0266-1/01 Use the latest version of ISO/IEC 10646

Ada should depend on the most recent versions of other Standards. There is some concern that referencing obsolete standards could hold up our standardization. So we should update to use the most recent version of 10646.

Should we update the references to the C and C++ standards? Yes, for the same reasons.

For C, there is a Corrigendum 2007 that has been added. [Editor's note: When checking this in December, there now is a new C Standard, 9899:2011, dated 2011-12-08! This is what our Standard will use in its references section.]

Someone says that C++ is in process and isn't finished yet. Tucker (looking on the web) says that it was "approved by ISO" in August. So we'll use the 2011 version of C++ as well.

We turn to the Note at the end of Ada.Wide_Characters.Handling.

Jean-Pierre suggests that we add an implementation-permission to use a newer (or older?) character set standard in a given Ada compiler. It should be implementation-defined which standard was used, so that implementations can use the standard that is appropriate for the host (or target).

Randy notes that would make it harder to write ACATS tests for character set issues (like case equivalence for identifiers).

Tucker suggests adding a parameterless function that returns a string representing the character set standard. A test could then depend on the character set version.

We also want a minimum amount of support, so we will require at least support for 10646:2003. That also will allow tests for commonly used characters (Greek, Cyrillic, etc.) since those characters are in every character set standard.

So add at end of 2.1:

Implementation Permission

> The categories defined above, as well as case mapping and folding, may be based on an implementation-defined version of ISO/IEC 10646 (2003 edition or later).
>
> AARM Ramification: This affects identifiers and packages Wide_Characters.Handling and Wide_Wide_Characters.Handling.

[Editor's Note: The AARM Note also should mention the result of 'Wide_Wide_Image, as that depends on case mapping and whether a character is considered a graphic character.]

Gary notes an error in the AI: Summary: Ada.Characters.Wide_Handling => Ada.Wide_Characters.Handling.

Another typo is noted in the !discussion “Adds an[d] Annex U...”

In A.3.5: Add a Note

> The results returned by these functions may depend on which particular version of the 10646 standard is supported by the implementation (see 2.1).

Add a function to A.3.5:

```
function Character_Set_Version return String;
```

> Returns an implementation-defined identifier that identifies the version of the character set standard that is used for categorizing characters by the implementation.

Add at the end of A.3.5:

Implementation Advice

> The string returned by Character_Set_Version should include either “10646:” or “Unicode”.

Approve AI with changes: 7-0-2.

AI05-0267-1/01 Improvements for aspect specification

For proposal (1), we agree with the recommendation in the AI that we make no change. While the proposed syntax is harder to remember (that is, write), it is easier to read because it doesn't put aspects in the middle of short declarations (like instantiations).

Considering proposal (2), the recommendation is that we should allow aspect_specifications on stubs. They should be allowed only if there is no specification, and not on the subunit body.

Randy notes that this is important: if you call a stub that has no separate specification, you will want see the Pre and Post (it is the caller's specification). It is silly to force a separate specification in this case only.

We agree that we should allow these on stubs. For the language-defined aspects, there cannot be a specification, and the (subunit) body cannot have any aspects.

Turning to proposal (3), Steve notes that precident is spelled wrong here.

We should allow implementation-defined aspects on everything that makes sense. So we should allow these on bodies without restriction.

13.3.1(34/3) will need to be updated. Gary thinks this should be a legality rule, because it is not clear otherwise. In addition, all generic formals should be included here.

13.3.1(34-35/3) should be moved to the end of the Legality Rules (after paragraph 16).

Considering proposal (4): It seems too late to add these new aspects now; most of this can be done with Assert pragmas.

It was suggested that we explicitly allow implementation-defined aspects to use 'Old and 'Result.

Straw vote on the full proposal (including proposal (4)): 2-6-1.

Should this be split into a Ada 2012 AI? Randy argues that we should split this off since we have already heard significant interest in this idea (Steve Michell). And it is easy to split. So create an Ada 2012 AI for this idea. [Editor's Note: This is AI12-0014-1.]

We need wording written for 13.3.1(34-35/3) to allow stubs in 34, and bodies and formals in 35. Ed will work with Randy on that.

Late Saturday, we return to this AI. Ed distributed his suggested wording (which will be Legality Rules moved after 13.3.1(16/3)).

Modify 13.3.1 (34)

> There are no language-defined aspects that may be specified on a renaming_declaration, a formal_type_declaration, or a subunit.

Modify 13.3.1 (35)

> An aspect shall not be specified in an aspect_specification given on a subprogram body that is a completion, or a subprogram_body_stub that is a completion.

We want 13.3.1(34) to cover all formal parameters, so change formal_type_declaration => generic_formal_parameter_declaration.

Add package_body, task_body, protected_body, and body_stub other than subprogram_body_stub to this wording.

In 13.3.1(35) “aspect” should be “language-defined aspect”. Erhard says that the subunit body is a completion of the stub. We look this up, and decide stubs aren't completed, they have their own rules. Thus "subunit" should be in 13.3.1(34).

The stub is a completion of a specification, so we talk about that in the completion rule, not the other rule.

The revised wording is:

Replace 13.3.1 (34/3) with the following, moving it after 13.3.1(16/3):

> There are no language-defined aspects that may be specified on a renaming_declaration, a generic_formal_parameter_declaration, a subunit, a package_body, a task_body, a protected_body, or a body_stub other than a subprogram_body_stub.

Replace 13.3.1 (35/3) with the following, moving it after 13.3.1(16/3):

> A language-defined aspect shall not be specified in an aspect_specification given on a subprogram_body or subprogram_body_stub that is a completion.

[Editor's note: I simplified this to eliminate the redundant "that is a completion". Hope no one thinks that hurts the readability.]

We are adding aspect_specification syntactically (before the **is**) to all bodies, and will add them at the end on the stubs.

Approve AI with changes: 8-0-1.

AI05-0268-1/01 Examples in new sections of the Standard

Gary suggests indenting the examples in the AI so they look better (especially in the formatted version).

(1) Tucker suggests making the parameter **aliased**, then we don't need the 'Access.

The examples should not use **limited private**, use instead:

```
type Container is tagged ...
```

The second example should have a different name: “Indexable_Container”.

We decide to call these “Barrel” rather than “Container”.

The objects are B and IB. Need a declaration of IB in the second example. Also need a declaration of Find.

Use “pear” in the second example. Drop the first see 4.1.5 in the second example (it will be self-contained).

For 5.5.2, use forward references to point to the container example.

> *Examples*
>
> For examples of use of generalized iterators, see A.18.32 and the corresponding container packages in A.18.2 and A.18.3.

In the AARM, add the Amazon link to John's book. :-)

Tucker has a typo in the draft Standard – in 13.12.1(6.1/3), the AI reference should be to AI05-0241-1, not AI05-0242-1.

Approve AI with changes: 8-0-0.

AI05-0272-1/00 Pragma and Attribute restrictions

[Editor's note: This discussion was based on e-mail, there was no empty AI, and the AI number was assigned after the meeting.]

Robert had suggested restrictions similar to the ones defined for aspects. This seems like a good idea.

What about attributes (**access**, **digits**, **range**...) and pragmas (**interface**) that are reserved words?

After much discussion, Tucker suggests that “identifiers specific to a pragma” be allowed to be reserved words (in 2.8(10)). “...is an identifier {or reserved word}...”

That seems to work well enough.

Straw poll on this solution: 6-0-3.

Jean-Pierre says that he doesn't like the keyword solution. But he doesn't offer an alternative.

Steve wonders about No_Use_Of_Pragma => Restrictions. He thinks this is illegal, as it violates itself.

Tucker will attempt to write this up.

Gary notes that this should disallow specification of attributes as well as uses of attributes.

AI05-0272-1/01 Pragma and Attribute restrictions

On Sunday, Tucker presents his draft AI.

[Editor's note: Tucker's draft did not include the wording change for 2.8(10) that was discussed the previous day. This was added to the final AI as some solution to this problem is needed.]

Steve would like to disallow Restrictions from No_Use_of_Pragma, because it is unclear what it means. Tucker thinks that is a silly thing to worry about.

Jean-Pierre says that it might be useful to say that there are no restrictions.

So add a AARM Ramification noting that No_Use_of_Pragma => Restrictions would necessarily be rejected (at compile-time or link-time), since it would apply to itself.

Add **mod** to the list of reserved words allowed for attributes here.

Randy notes that these apply to the run-time system. This leads to a discussion of why – eventually H.4(24) is found, which says all restrictions apply to the run-time.

This probably should be different. It's hard to imagine writing the run-time without Import (to take one example). But excepting the run-time might cause problems for some uses (if you want to get rid of pragma Priority, you probably want to do it everywhere).

There is a suggestion for a To Be Honest to say that these don't apply to the run-time. That seems pretty hand-wavy.

Tucker suggests that H.4(24) should only apply to restrictions that apply to the run-time behavior of the system. Steve says that No_Allocator would not be covered by H.4(24) in that case, and that should apply to the run-time. Jean-Pierre says Max_Tasks => 0 would also be a problem, that surely should apply to the run-time. So Tucker's idea won't work.

Randy asks if there is really any good reason not to add these restrictions to the exceptions in H.4(24). (Meaning that they would not apply to the run-time.) No one objects. So add No_Specification_of_Aspect, No_Use_of_Pragma, No_Use_of_Attribute to the list in H.4(24).

For pragma, add "..or the reserved word Interface."

Approve AI with changes: 7-0-0.

AI05-0273-1/00 Problems with the Old attribute

[Editor's note: This discussion was based on e-mail, there was no empty AI, and the AI number was assigned after the meeting.]

Steve had suggested that 'Old where the prefixed name depends on something that does not exist at the point that the subprogram is called is a problem. That's because the prefix of 'Old is evaluated when the subprogram is called, and all parts of the prefix have to make sense then. In particular, names/entities that are defined by the Post expression cannot be used in the prefix of 'Old.

This meets general agreement.

In Steve's normal way, he continues by throwing out other questions. In `X(I'Old)'Old`, the order that the constants are evaluated matters. We don't have any wording to deal with cases like this.

If F is a function that increments I, then in

```
F'Old + I'Old
```

the value of I is not that on entry. We need a TBH (at least) to explain.

Steve will work on the ordering thing, and the restrictions on the name.

We should have user notes about 'Old (22.c/3 could be a starting point), and preconditions/postconditions with side-effects.

Erhard would like such preconditions/postconditions to be a bounded error. That sounds harder, because we can't then talk about "side-effects" in an unspecified way. (This discussion gradually separated into what became AI05-0274-1. Continue following it there.)

Later Friday, Steve sends a rewrite of 'Old:

X'Old

> For each X'Old in a postcondition expression that is enabled, a constant is implicitly declared at the beginning of the subprogram or entry. The constant is of the type of X and is initialized to the result of evaluating X (as an expression) at the point of the constant declaration. The value of X'Old in the postcondition expression is the value of this constant; the type of X'Old is the type of X. These implicit constant declarations occur in an arbitrary order, except that if one Old attribute use occurs as part of the prefix of another, then the constant declaration associated with the first use precedes that of the second.
>
> Use of this attribute is only allowed within a postcondition expression. The prefix of an Old attribute shall not contain either a Result attribute use or a use of a declaration declared within an enclosing expression but not within the given Old attribute prefix (for example, the loop parameter of an enclosing quantified_expression).

Steve uses this wording so that:

```
T'(for all I in A'range => A(I) /= 0)'Old -- Legal

(for all I in A'range => A(I) /= A(I)'Old) - Illegal
```

"attribute use" should be "attribute_reference" in all places. So:

X'Old

> For each X'Old in a postcondition expression that is enabled, a constant is implicitly declared at the beginning of the subprogram or entry. The constant is of the type of X and is initialized to the result of evaluating X (as an expression) at the point of the constant declaration. The value of X'Old in the postcondition expression is the value of this constant; the type of X'Old is the type of X. These implicit constant declarations occur in an arbitrary order, except that if one Old attribute_reference occurs as part of the prefix of another, then the constant declaration associated with the first reference precedes that of the second.
>
> Use of this attribute is only allowed within a postcondition expression. The prefix of an Old attribute shall not contain either a Result attribute_reference or a use of a declaration declared within an enclosing expression but not within the given Old attribute prefix (for example, the loop parameter of an enclosing quantified_expression).

The discussion veered back to AI05-0274-1 at this point. After quite a lot of discussion, we returned to wordsmithing Steve's wording. We make a few more changes and end up with:

X'Old

> For each X'Old in a postcondition expression that is enabled, a constant is implicitly declared at the beginning of the subprogram or entry. The constant is of the type of X and is initialized to the result of evaluating X (as an expression) at the point of the constant declaration. The value of X'Old in the postcondition expression is the value of this constant; the type of X'Old is the type of X. These implicit

> constant declarations occur in an arbitrary order, except that if one Old attribute_reference occurs as part of the prefix of another, then the constant declaration associated with the first reference precedes that of the second.
>
> Reference to this attribute is only allowed within a postcondition expression. The prefix of an Old attribute_reference shall contain neither a Result attribute_reference nor a use of an entity declared within the postcondition expression but not within prefix itself (for example, the loop parameter of an enclosing quantified_expression).

We discuss the A(I'Old)'Old example. This means the same as A(I)'Old, do we really need to allow both? Randy argues that the form with two 'Old attributes is clearer, because it makes it clear that I is not the current value. Tucker says that implies the user doesn't understand the model. Randy thinks that hardly anyone is going to understand the nuances of the model. Others think that having two ways to write the same thing is just as confusing; readers are going to be looking for a difference that does not exist. The group goes with this second opinion. So we shouldn't allow 'Old in the prefix, because then we can get rid of the ordering business.

X'Old

> For each X'Old in a postcondition expression that is enabled, a constant is implicitly declared at the beginning of the subprogram or entry. The constant is of the type of X and is initialized to the result of evaluating X (as an expression) at the point of the constant declaration. The value of X'Old in the postcondition expression is the value of this constant; the type of X'Old is the type of X. These implicit constant declarations occur in an arbitrary order.
>
> Reference to this attribute is only allowed within a postcondition expression. The prefix of an Old attribute_reference shall not contain a Result attribute_reference, nor an Old attribute_reference, nor a use of an entity declared within the postcondition expression but not within prefix itself (for example, the loop parameter of an enclosing quantified_expression).

We return to discussing AI05-0274-1.

AI05-0273-1/01 Problems with the Old attribute

The discussion of AI05-0274-1 abruptly turns back to 'Old when Jean-Pierre posts an example:

```
Post => I > 0 and then Tab(I)'Old = 1
```

This is trouble, because the Tab(I) is (unconditionally) evaluated on entry; if I is 0, this will blow up. (We're assuming that Tab is an array starting at 1.)

There is no obvious solution to this; if I can change during the execution of the subprogram (and the compiler usually has to assume that), the evaluation of 'Old has to be unconditional. There is no way for the compiler to know if the reference to 'Old will be needed, and it had better have the value if it is needed. Thus the compiler has to declare the constant and evaluate the prefix. Clearly this will allow you to shoot yourself in the foot.

Tucker and Steve are arguing whether or not we need to evaluate 'Old in examples like this, if the compiler can prove that it does not need the result. We surely don't want to require compilers to evaluate these only for the side-effects that we hope they don't have.

Tucker wants such a rule for all pre and postconditions.

Steve suggests an Implementation Permission for 'Old:

If the value of an Old attribute_reference is not needed, then the implementation is permitted to omit the declaration of the corresponding implicitly-declared constant.

This doesn't go far enough; it should be possible to omit any function call that is not needed.

Randy suggests that this permission should be tied to Assertion_Policy, and apply to all expressions controlled by the policy. Steve suggests the permission be modeled on the Pure function permission.

Erhard will take this and try to create a permission for 11.4.2. This pushes this rule into AI05-0274-1, which is about general rules applying to all assertions. Follow the rest of the discussion there. [Sorry, but these discussions are inseparably intertwined, yet somehow ended up with two separate AIs. It's the best I could do – Editor.]

We go back to discussing AI05-0274-1, and then it is time for dinner. So we don't return to this until Saturday morning. We return to thinking about 'Old in a conditionally executed expression.

Tucker suggests banning 'Old in part of an expression that does not always get executed.

But Jean-Pierre objects that this is perfectly reasonable and harmless:

```
Post => (I > 0 and then (Tab'Old(I) = Tab(I))
```

Randy suggests to apply the rule only to dynamic things (allowing simple names and selected components not in a variant); Tucker suggests “statically denotes”. Steve worries that this would toss out useful stuff. Not really, it would usually require putting the 'Old on the outer object (not indexing or components of it).

Jean-Pierre is concerned that this rule is pretty restrictive. He suggests deferring the exception if the expression is not always evaluated. In that case, the exception would be deferred until the 'Old is evaluated as part of the postcondition expression (and if it never is, the exception is not raised). That would be a pain to implement (every 'Old reference would have to be wrapped in a handler in case an exception is propagated).

There is some sentiment for restricting the prefix generally to simple names and selected components that are not in variants (probably not discriminant dependent). That is, the restriction would apply to all uses of 'Old.

Tucker wonders if there is enough value to even allowing selected components and the prefix of 'Old. Steve suggests that if the object is controlled with an integer component, copying it can be expensive.

The idea is to only allow names that cannot fail a check.

This would eliminate problems with confusion about the difference between T'Old(I) and T(I)'Old. It also would eliminate any problem with short circuit evaluation.

Jean-Pierre says this would disallow using a postcondition to check a time-budget.

```
Post => Clock - Clock'Old <= 1.0
```

This says that if the function runs too slow, raise an exception *after it completes***.** That seems exactly wrong thing to do.

Steve notes that the proposed restriction disallows Selector_Function(X)'Old. Erhard notes that Selector_Function(X'Old) is what you really mean, and it's what you should say.

It's clear that we could allow more in the future, if there is a demonstrated need. The important problem is not to allow too much now.

With function cases that have side-effects, F(P'Old) might not be the same a F(P)'Old. But those are considered bad anyway.

We haven't identified any real problems with a general rule, but we are still worried that we're throwing out too much. There could be quite a run-time cost to this rule. For instance:

```
Post => Giant_Array_of_Integers(An_In_Parameter)'Old >= Func'Result
```

would be illegal. You could write:

```
Post => Giant_Array_of_Integers'Old(An_In_Parameter) >= Func'Result
```

which would be legal, but it would require copying the entire giant array instead of a single element.

Implementations could optimize out the extra copies in many cases, but that seems like a lot of work and it's hard to say if implementations will actually do that.

Tucker suggests being restrictive only in the case of an expression that might be not evaluated: in a *dependent*_expression, or other than the first operand of a short circuit. Then we would require a simple name as the prefix.

Simple name isn't defined. direct_name is just identifier. So we need "direct_name or expanded name". Tucker wonders whether "usage name" would work, but it seems to include record components with arbitrary prefixes.

Someone suggests "a name that statically denotes", as Tucker originally proposed a half hour ago. We look up the definition of "statically denotes" in 4.9(14), it seems to have the right meaning.

Tucker suggests using 4.9(32.1-6/3) to get ideas. We can't use exactly that wording, because it assumes that the expression is static. But we can write something like it. The following is proposed:

A subexpression is potentially unevaluated if occurs within:

- any part of a if_expression other than the first condition;
- any *dependent*_expression of a case_expression;
- the right-hand side of a short-circuit operation;
- a membership_choice other than the first of a membership operation.

The prefix of an Old attribute_reference that is potentially unevaluated shall statically denote an entity.

This looks right. [Editor's note: Not quite. The term subexpression is not defined or used in the normative wording of the Standard. We have to say "An expression" instead, not quite as clear, but correct.]

Moving on. Steve has another question: what about R.Access_Disc'Old or Acc_Param'Old? The type of the expression is the type of the object, that doesn't work for anonymous access types (which object has a unique type).

Let's ignore this, especially as both of these prefixes are constants, so why would you use 'Old? And it would have the right accessibility if it was a new object subtype anyway. No one is going to be confused about the type. So this is too pedantic to care.

Make minor corrections to the wording:

A subexpression is potentially unevaluated if it occurs within:

- any part of a if_expression other than the first condition;
- a *dependent*_expression of a case_expression;
- the right operand of a short-circuit control form; or
- a membership_choice other than the first of a membership operation.

The prefix of an Old attribute_reference that is potentially unevaluated shall statically denote an entity.

Approve AI with changes: 9-0-0.

AI05-0274-1/01 Side effects during assertion evaluation

[Editor's note: This discussion gradually forked from the AI05-0273-1 discussion, the AI was created during the meeting, and the AI number was assigned after the meeting.]

Erhard sent a note with proposed wording [Editor's note: this is now the above-mentioned AI version]:

> It is a bounded error in a pre- or postcondition to invoke a subprogram that alters the value of a variable that is not locally declared in the subprogram. If the bounded error is detected, Program_Error is raised. If not detected, execution proceeds normally, but values denoted by X'Old for any prefix X might not denote the value denoted by X at the beginning of the subprogram with the affected pre- or postcondition.

Randy is concerned that it calls out a particular odd effect and ignores any others. Erhard and others claim that 'Old is the only truly weird side-effect.

Steve worries that this gives the compiler permission to do the wrong thing. That is not the intent. It should be like a note. Use “might”, and this should talk about a subprogram call. So we get:

> It is a bounded error in a pre- or postcondition to invoke a subprogram that alters the value of variable that is not locally declared in the subprogram. If the bounded error is detected, Program_Error is raised. If not detected, execution proceeds normally.
>
> As a consequence, values denoted by X'Old for any prefix X might not denote the value denoted by X at the beginning of the subprogram call with the affected pre- or postcondition.

After a lengthy discussion of issues specific to 'Old (see AI05-0273-1 for that), we return to the bounded error.

Straw poll on whether this should be a Bounded Error: 4-4-1.

The biggest objection is that side-effects that don't matter would be banned here (such as a memo function that saves previous results). [Robert Dewar complained about this in ARG e-mail around this time, responding to Erhard's first attempt at this AI - Editor.]

Tucker suggests making this a bounded error that a compiler couldn't detect. That is, this is a bounded error only if the result could change as a result of this side-effect. That gets more support; Erhard will try again.

We then return to discussing AI05-0273-1. After quite a bit of discussion on that, Erhard sends a new proposal.

> It is a bounded error in a pre- or postcondition to invoke a subprogram that alters the value of any variable that is not locally declared in the subprogram so that the value of the pre- or postcondition expression is affected by the altered variable. If the bounded error is detected, Program_Error is raised. If the bounded error is not detected, values denoted by X'Old for any prefix X might not denote the value denoted by X at the beginning of the subprogram call with the affected pre- or postcondition.

For some reason, we've gone back to just talking about 'Old. That's not what we want, if we want this at all.

> It is a bounded error in a pre- or postcondition to invoke a subprogram that alters the value of any variable that is not locally declared in the subprogram so that the value of the pre- or postcondition expression is affected by the altered variable. If the bounded error is detected, Program_Error is raised. If the bounded error is not detected, execution proceeds normally. As a consequence, values denoted by X'Old for any prefix X might not denote the value denoted by X at the beginning of the subprogram call with the affected pre- or postcondition.

Tucker suggests that this should apply if it changes the value of any pre- or postcondition of the same subprogram. We want to be able to assume that if the pre- or postconditions are True after they get evaluated, the value should not change.

We get side-tracked again by an example that Jean-Pierre introduces. Again, see AI05-0273-1 for that discussion.

Back to Erhard's last bounded error text:

Tucker would like to simplify. Also, this wording should apply to all Assertion_Policy things. Erhard would like to move the other bounded error here as well.

It's time for dinner, we'll let Erhard take a stab at this.

AI05-0274-1/02 Side effects during assertion evaluation

On Saturday, we look at Erhard's latest draft:

Invariants need to be included; “any type_invariants that are evaluated on the exit of the subprogram”.

The third question should be “bounded errors in assertions? (Yes).

Subtype predicates need to be added to the list of things that are “assertions”. Tucker suggests putting them in RM order so that the (see xxx) are in order.

“Boolean” should be “boolean” here.

Remove hyphen from “run-time”.

> It is a bounded error in an assertion expression that is evaluated as part of a call of or return from a callable entity C to alter the value of any variable that affects the value of any other such assertion expression. If the bounded error is ...

Steve worries about “affects” here. He wonders how you would tell if it is OK to raise Program_Error.

An example of a questionable situation:

```
Incb(X) and X > 0
```

The value is only affected if X = 0. [Editor's note: This specific expression is almost certainly illegal, because the function Incb has to have an **in out** parameter in order to modify X (X is passed by copy), and the other use of X in this expression would trigger the unspecified order rules of 6.4.1 (specifically violating 6.4.1(6.18/3)). But it would be easy to write a similar example that is legal.]

Erhard suggests “could affect”. Tucker wonders if there is a more technical way to put this.

He suggests it alters a variable that appears in the expression.

Tucker suggests “that could alter the value of the expression on an immediate subsequent evaluation”.

The permission should say “omit the call of a function”.

Randy notes that this is inconsistent with Ada 2005 for pragma Assert. Existing programs that currently work as expected could now raise Program_Error.

Tucker says the more incompatible the better in this case; he thinks the programmer is doing something unbearably tricky if they have a problem. Compilers are unlikely to be able to detect this Bounded Error except in the worst cases.

This is a new AI05 AI:

Approve AI with changes: 6-1-2.

Steve opposed because he thinks this notion of “affects” is too imprecise.

AI05-0274-1/03 Side effects during assertion evaluation

After lunch on Saturday, we look at Erhard's final version of the AI.

Change the end of the the second bounded error:

...but the assertion expression need not hold if immediately reevaluated.

Delete the following note.

!discussion

...apply {to} the pragma...

Gary asks to remove hyphens from side-effects.

Reapprove with editorial changes: 6-1-2. [Same objections as before.]

AI05-0276-1/01 Violation of predicate legality rules in generic units

[Editor's note: This AI was created from an e-mail sent during the meeting, with the AI number being assigned after the meeting.]

John had sent a question to the ARG list which was adequately answered. But in answering that, Steve asked where the Program_Error would be raised in 3.2.4(23).

We discuss this for a while. The best answer seems to depend on the implementation techniques. We think that the best plan is an implementation-permission to allow it to be raised either at the uses or during the elaboration of the instantiation. Steve will write this permission.

Later on Saturday, we look at the wording Steve proposed (via e-mail).

Tucker suggests saying “at some {earlier} point during the elaboration of the instance”.

Gary wonders if this should apply to other rules that are like this (raise Program_Error at the instance for the failure of a Legality Check at run time). Making this a blanket rule is hard, because it only applies in a few cases.

We are using the single paragraph version of this rule.

Approve intent: 4-0-5.

We discuss why there isn't stronger approval of this AI. Several people say that they don't like the indecision; it doesn't seem that it buys much. They especially don't like "at some point during the elaboration".

If we only go with a single option, which one should we chose? Randy would prefer that the exception be raised at the point of the violation. It's easier to implement in his shared generic implementation (the instance does not know the contents of the body). He also notes that it is consistent with the way that accessibility checks are handled.

The group agrees with his logic. So just add ”...at the point of the violation.” at the end of the existing wording.

Make this a new AI05.

Approve AI with changes: 8-0-1.

AI05-0277-1/01 Aliased views of extended return objects

[Editor's note: This AI was created from an e-mail sent in June, with the AI number being assigned after the meeting.]

Ed had noted that there is a compatibility problem with dropping the keyword **aliased** from extended return statements. He is proposing that we allow the keyword, and then have a rule like:

If the keyword **aliased** is present, the type of the extended return shall be immutably limited.

Tucker argues that we should revert to the rule that you have to write **aliased** for it to be aliased. Is there a compatibility problem with Ada 2005? No, because even if you gave **aliased**, it was not aliased by 3.10(9/2)!!

So we decide to follow Tucker's suggestion. However, writing that will be hard, because there is no declaration to talk about. Tucker suggests just changing the syntax of extended_return_statement to fix this problem.

So we decide to do this: remove the ", as the return object of an extended_return_statement..." from 3.10(9/3), add ", extended_return_object_declaration" to the list of things that are aliased if **aliased** appears.

Replace 6.5(2.1/3) by:

```
extended_return_object_declaration ::=
  defining_identifier : [aliased][constant] return_subtype_indication [:= expression]

extended_return_statement ::=
  return extended_return_object_declaration [do
    handled_sequence_of_statements
  end return];
```

Add a legality rule after 6.5(5.8/3):

If the keyword **aliased** is present in an extended_return_object_declaration, the type of the extended return object shall be immutably limited.

Randy will check if there are any rules (added for Adam) that were added specifically because this was not a _declaration. Now it is. [Editor's note: Randy was thinking of AI05-0205-1. Its change is no longer necessary.]

Approve AI with changes: 8-0-1.

AI05-0278-1/01 Set_CPU called during a protected action

[Editor's note: This AI was created from an e-mail sent in September, with the AI number being assigned after the meeting.]

Randy worries that the proposed semantics is distributed overhead – you would have to check after every protected action whether Set_CPU was called. Actually calling Set_CPU would be very rare, yet we would have a non-zero cost in something that is common and which should be as cheap as possible.

Tucker suggests that Set_CPU should only be effective when the task reaches a task dispatching point, that is when it is rescheduled.

Set_Priority is not a task dispatching point, so this should not be one either.

But we worry that this is too much change; IRTAW may have had a reason for choosing this semantics.

So replace the last sentences of D.16.1(27/3) with:

A call of Set_CPU is a task dispatching point for task T unless T is inside of a protected action, in which case the effect on task T is delayed until its next task dispatching point. If T is the Current_Task, the effect is immediate if T is not inside a protected action, otherwise the effect is as soon as practical.

Similar wording should be used for Assign_Task (see D.16.1(26/3)).

Approve AI with changes: 6-0-3.

AI05-0279-1/01 Renaming an invalid Unchecked_Conversion result

[Editor's note: This AI was created from an e-mail sent in September, with the AI number being assigned after the meeting.]

We should do this, because we do not want renames to be less capable than assignment. Steve will write wording for this tonight if we don't have too much wine at dinner.

On Sunday, we look at Steve's wording proposal.

> A call to an imported function or an instance of Unchecked_Conversion is erroneous if the result is scalar, and the result object has an invalid representation, and the result is used other than as the expression of an assignment_statement or an object_declaration.[as the name of an object_renaming-declaration,] or as the prefix of a Valid attribute. If such a result object is used as the source of an assignment, and the assigned value is an invalid representation for the target of the assignment, then any use of the target object prior to a further assignment to the target object, other than as the prefix of a Valid attribute reference, is erroneous.

Steve confused the uses of curly (insertions) and square (deletion) brackets. So the square brackets enclose the new wording.

The period in front of the square bracket should be a comma.

Replace "as the name of an object_renaming-declaration" with "as the *object*_name of an object_renaming_declaration", fixing an error in the process.

Approve AI with changes: 7-0-0.

AI05-0284-1/00 Accessibility of anonymous access returns

[Editor's note: This AI was created from an e-mail sent in August, with the AI number being assigned after the meeting.]

We spend a while trying to figure out what Steve thinks the problem is in his mail.

It's not clear whether Steve wants more or less accessibility. Steve himself is no help. We table this until someone understands the problem.

Detailed Review of Ada 2012 AIs

AI12-0006-1/01 Accessibility of null

3.10.2(12.2/3) says "(library-level if null)". There are several other places that say this as well. Probably that is what we need to do for 3.10.2(13/2) is add to the end:

(or library-level if the actual is null)

Promote this to an AI05. [AI05-0270-1 – Editor.]

Approve AI with changes: 7-0-2.

AI12-0007-1/01 Accessibility of access discriminants of a subtype

Tucker says that Steve's interpretation is wrong, because the *check* is occurring in a return statement. He says that the bullet 3.10.2(12.1/2) applies, because the subtype_indication here is not necessarily the same as the one from the ones mentioned in 3.10.2(12/2).

That's confusing, the natural interpretation is that the subtype_indication mentioned in the lead-in and in the bullet is the same one. We should add something to clarify.

Add an AARM note after 3.10.2(12.d/2), mentioning:

Note that the constrained subtype might have been defined in an earlier declaration (as a named subtype).

So write this as a Ramification, and add the above.

Promote this to an AI05. [AI05-0281-1 – Editor.]

Approve AI with changes: 5-0-4.

AI12-0008-1/01 Bad ancestor constraints for extension aggregates

Tucker wonders if this type is really constrained. If not, the aggregate would be OK, and the assignment would make the check (and fail).

Steve does not want that, because then you would allow equality checks on such aggregates. That would have a distributed overhead – record representations would have to assume that they could be unconstrained even if the first subtype is constrained. Tucker is convinced.

4.3.2(8) does not apply because the discriminants are inherited.

4.3.2(7) should be fixed. The constraints that apply to the ancestor should be checked. Or something like that.

Steve is assigned to come up with wording.

On Sunday, Steve presents his suggested wording changes. [Editor's note: See AI05-0282-1/02 for this wording.]

Steve says that a lot of information is missing from the existing wording. In particular, where the discriminant values come from is not specified.

```
type T1 (D : integer) is tagged private;

type T2 is new T1(123) with ...;

X : T2 := (T1 with ...); -- Should be legal, can't specify the discriminant explicitly.
```

Tucker claims that the discriminants are inherited for T2, so D has to be specified in the aggregate (and it better have the value 123, or Constraint_Error should be raised). Thus the comment in the example above is wrong. The example should be instead:

```
type T2(D2: Integer) is new T1(123) with ...;

X : T2 := (T1 with D2 => ...); -- Should be legal, can't specify D explicitly.
```

But the old wording talks about initializing the aggregate based on the ancestor type (which is unconstrained and does not supply the value of D).

Tucker claims this is a general issue, we generally don't talk about those discriminants. He claims that 3.7(18) says that such discriminants are “specified”. That means that they aren't components at all, and they **just have** the correct values.

This is a lot of hand-waving, but we aren't going to fix that. Steve says we then don't need most of the wording changes in paragraph 7.

Tucker says he would like to change “ancestor type” to “subtype denoted by the subtype_mark”. Upon more thought, he's not sure.

We turn to the important change to paragraph 8. Steve just eliminated the part about inherited discriminants, because we don't really care where they come from.

Steve notes that there are 6 cases to worry about.

- For each discriminant, it is either given by the aggregate part or the ancestor part.
- There are three possible forms of the ancestor part (an expression, a constrained subtype_mark, or an unconstrainded subtype_mark).

Tucker worries about the wording “corresponding discriminant”. Is that appropriate for an inherited discriminant?

```
type Root (D : Integer) is tagged ...
subtype S1 is Root(1);
subtype S2 is Root(2);

type Ext is new S1 with ...
Var : Ext := (S2 with ...);
```

Tucker says that “corresponding discriminant” is not defined for an inherited discriminant.

He admits that a discriminant could correspond to itself.

He suggests adding “if any”. Because the type of the aggregate might not have any constraints:

```
type Ext2 is new Root with null record;

Var : Ext2 := (S2 with null);
```

In which case there is nothing to check against.

So use Steve's paragraph 8:

> If the type of the ancestor_part has discriminants and the ancestor_part is not a subtype_mark that denotes an unconstrained subtype, then a check is made that each discriminant of the ancestor has the value specified for a corresponding discriminant, either in the record_component_association_list, or in the derived_type_definition for some ancestor of the type of the extension_aggregate. Constraint_Error is raised if this check fails.

Make the following improvements to the above:

“...corresponding discriminant, {if any,} ...”

“each discriminant {determined by}[of] the ancestor{_part}...”

Promote this to an AI05. [AI05-0282-1 – Editor.]

Approve AI (as AI05) with changes: 6-0-2.

AI12-0009-1/01 Iterators for Directories and Environment_Variables

This does not have a simple solution. We discuss options briefly. Tucker suggests that because of that, we should leave this as an AI12.

AI12-0010-1/01 Stream_IO should be preelaborated

Tucker suggests that we poll implementors as to whether Streams.Stream_IO could be preelaborated.

Randy notes that we polled implementers in 2003 or 4 about this. GNAT and Janus/Ada would have had minor effort to do this.

Tucker says that AdaMagic would be able to make the change without problem.

Gary will take an action item to check this in GNAT.

The feeling is that if we can do this now, we should, as it is a significant hardship for distributed systems (and makes it much less likely that code can be preelaborated).

Randy should send a query to the ACAA mailing list (as well as the ARG list) to ask other implementers for their opinion.

Tucker reports that he's already made the change in AdaMagic. (The wonders of the internet at work...)

Gary reports that he got Stream_IO to compile for GNAT by adding Preelaborate to it and a couple of other packages. So probably GNAT will not have a problem.

Promote this to an AI05 AI. [AI05-0283-1 – Editor.]

We'll leave this open for the next meeting, so we can get feedback from other implementers.

AI12-0011-1/01 Behavior of Random.Reset

What should this do? Pretty much anything could be inconsistent, in that the representation might have to change. Raising an exception also would be inconsistent.

Tucker suggests (after several false starts) that the State be required to be initialized to the default initial state (as described in A.5.2(28)). That might require adding default values, but even if this is an array, in Ada 2012 they could use Default_Initial_Value to set this. So no representation change should be needed.

So, in A.5.2(29), we should add: “The default initial value of an object of State corresponds to the default initial value of all generators.”

[Editor's note: We can simplify this wording by talking about the implicit initial value of the (sub)type State, that's more consistent with the previous paragraph. Specifically: “The implicit initial value of type State corresponds to the implicit initial value of all generators.”

Promote this to an AI05. [AI05-0280-1 – Editor.]

Approve AI with changes: 9-0-0.

AI12-0012-1/01 Failure behavior of Directories.Create_Path

Tucker suggests following the model of Delete_Tree. So add to the end of A.16(61/2):

If Use_Error is propagated, it is unspecified whether a portion of the directory path is created.

A similar question is raised about Copy_File. Tucker suggests something similar in A.16(69/2):

If Use_Error is propagated, it is unspecified whether a portion of the file is copied.

Promote this to an AI05. [AI05-0271-1 – Editor.]

Approve AI with changes: 7-0-2.

AI12-0013-1/01 More issues with the definition of volatile

The problem is that accessing a volatile variable is not a synchronization point. So it requires some other method for synchronization. [Editor's note: formally, this is that accessing a volatile variable is not "sequential".]

It is suggested that it would be simpler to simply change "volatile" to "atomic" in C.6(16/3).

Tucker worries that making this change destroys the point. This has nothing to do with sequentiality; partial updates are allowed. But those partial updates come in the same order. If a volatile object has two parts, then it cannot be the case that one task sees the first part updated and not the second; and another task sees the second part updated but not the first. But any other order would be OK.

We don't want to change the meaning of C.6(16/3). But we do need to make it less misleading. We probably should add Geert's note to C.6(16/3).

The Implementation Note probably is overly pessimistic. Randy notes that sequentiality could occur in many ways other than Atomic (protected actions, the various ways you can get signalling), and in those cases, you might need barrier(s) for volatile objects. But you don't need such barriers if there is nothing that is sequential.

We send a note to Alan to find out if we understand the problem correctly.

On Saturday, Ed notes that Alan replied to this question. He mainly wondered why Atomic and Volatile are different, he thinks they both should be sequential. That's pretty obvious: atomic objects have to be read/written indivisibly (so only a few types can be used for atomic objects, and thus ensuring sequentiality is cheap), volatile has no such requirement (so any type can be used for a volatile object and ensuring sequential execution is not cheap). We would not want to force locking for every volatile object (that would be quite expensive).

So we'll stick with yesterday's resolution.

Tucker would like better wording for the note to be added to C.6(16/3):

To avoid erroneous execution, other mechanisms should be used to ensure sequential access (9.10) to volatile variables.

That's not well-liked. Take 2:

A use of an atomic variable or other mechanism may be necessary to avoid erroneous execution and to ensure that access to volatile variables is sequential (see 9.10).

C.6(16.a): “should involve” should be “may require”. And change to atomic as Geert suggests.

Randy wonders if we might need barriers for volatile if the sequential execution is caused by some other item (protected actions, for instance). Tucker says we probably would, but we can't define that in detail, and thus we won't mention it in the AARM note.

The new AI should give the correct example and note that the example in AI05-0117-1 is wrong.

Promote this to an AI05. [AI05-0275-1 – Editor.]

Approve AI with changes: 7-0-2.

AI12-0015-1/00 Ada unit information

AI95-0282-1 was an old proposal. We discussed this and suggested that attributes were more appropriate than magic subprograms.

Randy wants to know whether we are interested in this idea for the future. The group thinks that we might want to consider this for Ada 2020. So we'll keep this AI around (as an AI12).

AI12-0016-1/00 Implementation model of dynamic accessibility checking

Is this problem still open? Steve says that it is. We will need to decide if this is a language problem or an implementation problem. At the very least, we need to update the AARM to say that the small integer model does not work.

We don't want to try to deal with this now; we need more implementation experience. But there certainly should be an AI12 on this topic, at the very least to fix the AARM notes.