Minutes of the 16th ARG Meeting

21-23 June 2002

Vienna Austria

Attendees: Steve Baird, John Barnes, Randy Brukardt, Alan Burns (Friday and Saturday morning), Kiyoshi Ishihata, Pascal Leroy, Stephen Michell (Friday and Saturday), Erhard Ploedereder, Jean-Pierre Rosen, Tucker Taft

Observers: Tullio Vardanega (Friday and Saturday)

Meeting Summary

The meeting convened on 21 June 2002 at 14:30 hours and adjourned at 15:30 hours on 23 June 2002. The meeting was held in a conference room at the Parkhotel Schönbrunn on Friday and Saturday, and in a room at the Institut für Rechnergestützte Automation of the Technische Universität on Sunday, both in Vienna, Austria.

The meeting covered the entire agenda for amendment AIs, and a few normal AIs. The majority of the meeting time was spent on amendment AIs, with a few hours on the third day spent on normal AIs. Vienna had an unusual heat wave during the meeting, and the meeting rooms were very warm.

Once again, while there was no official vote, I (Randy) think that I speak for others in thanking Pascal and the Ada Europe organizers for the facilities and refreshments.

Meeting Minutes

There were no comments on the minutes of the 15th ARG meeting. The minutes were approved by acclamation.

Next Meeting

The next ARG meeting is set for October 11-13, 2002, hosted by Tucker Taft in the Boston area.

A discussion of the following meeting ensues. SigAda (in December) is too soon after the October meeting. A meeting date of February 7-9, 2003 is tentatively selected. A host is needed. Several people in cold weather cities volunteer. A warm weather location is needed. Joyce Tokar in Phoenix, AZ, is the obvious victim, umm, candidate. Pascal will contact Joyce to make arrangements if possible.

Old Action Items

The old action items were reviewed. Following is a list of items completed (other items remain open):

Steve Baird:

- AI-216
- AI-251
- AI-280
- AI-291 (Study the recommended level of support in chapter 13 to find any problems with aliased and by-reference types. If any are found, create an AI to correct them.)

Randy Brukardt:

- AI-224
- AI-248
- AI-259

- AI-279
- AI-283

Editorial changes only:

- AI-85
- AI-147
- AI-246
- AI-254
- AI-260
- AI-262

Gary Dismukes:

- AI-196

Pascal Leroy:

- AI-228
- AI-284

Tucker Taft:

- AI-217-04
- AI-266
- AI-286
- AI-288
- Funding: Tried to track down funding for WG9 and ARG, handed one possibility over to Jim Moore to pursue.

Joyce Tokar:

- AI-249

New Action Items

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

Steve Baird:

- AI-251
- AI-295

John Barnes:

- AI-296

Randy Brukardt:

- AI-224
- AI-292
- AI-299

- AI-301
- AI to fix the contract model violation of C.3.1(8) (new AI-303)
- Object_Size attribute

Editorial changes only:

- AI-216
- AI-217-04
- AI-262
- AI-276
- AI-284

Alan Burns:

- AI-249
- AI-265
- AI-266-02
- AI-297
- AI-298

Gary Dismukes:

- AI-158

Bob Duff:

- AI-239
- AI-287 (split into aggregate part and constructor function part)
- Be the test creator of last resort

Mike Kamrad:

- 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-264
- AI-285
- Contact Joyce Tokar to see if the February meeting can be held in Phoenix, AZ.

Steve Michell:

- AI-148
- AI-250

Erhard Ploedereder:

- AI-237

Tucker Taft:

- AI-133
- AI-162
- AI-167
- AI-188
- AI-214
- AI-230
- AI-231
- AI-252
- AI-270
- AI-282
- AI-286
- AI-288 (split into two AIs: pre/postconditions and invariants).
- AI-290
- AI-293
- Physical units (length/dimensional analysis), whereby square meters are generated by result of multiplying meters; subtypes with special attributes would be used to provide the specifications of these dimensions, reducing the introductions of lots of extra operators.
- Make a proposal for partial parameter lists in generic formal packages.

Joyce Tokar:

- CPU time (separate from real-time) [jointly with Mike Kamrad]

Mike Yoder:

- 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

Detailed Review

The minutes for the detailed review of AIs are divided into existing amendment AIs and non-amendment AIs. The AIs 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 six votes for, one vote against, and two abstentions.

Detailed Review of Amendment AIs

AI-216 - Unchecked Unions -- Variant Records With No Run-Time Discriminant

Steve Baird gives a short description of the minor changes to the AI.

"Said to be" is not proper RM language. Change it to "called" in the first paragraph where it occurs (two occurrences) and to "defined to have" in the second paragraph where it occurs.

Tucker queries why "Size" is under "static semantics". Steve Baird replies that Size is related to representation items. Tucker agrees to leave it alone.

Remove the quotes around "inferable discriminants" and italicize the first occurrence. Make the list a set of bullets.

Approve AI with changes: 6-0-3.

AI-217-04 Handling mutually recursive types via type stubs with package specifiers

Tucker explains the wording.

Erhard notes that he doesn't like 3.10.1(3.1) "…require that the completion exist." Exist is not a well-defined term. Is there a check, and where will it occur? Tucker replies that there is a check. This is an introductory sentence and the rules are in paragraphs 5 through 10. The second sentence of this paragraph should be in square brackets in the AARM. After discussion, Erhard decided that the wording is OK as it is.

Discussion turns to the completion rule. Erhard was concerned about visibility, but is convinced that is not is an issue.

Someone wonders if there is a problem with **private with**? No one is immediately sure. Steve Baird suggests that breaking privateness is not a problem here, as this is just a semantic dependence. So no change to the rule is needed.

Discussion continues on to the dereference rule. Someone comments that this is very confusing. Tucker admits that it is confusing to him, too.

Erhard wonders how E can ever match D'Class (where D and E are the names used in the wording of the AI). We look at an example:

```
package Q is
    type T is tagged separate in P;
    type A is access T'Class;

    O : A;
end Q;

package P is
     type T is ...
     OC : T;
```

```
        ...

        OC := O.all;
    end P;
```

In this example, the expected type of O.all is Q.T'Class which is D'Class; the type of OC is P.T which is E.

Tucker notes that there is an assumption that the completion of the class-wide type is the same as the class-wide type of the completion (this should probably be stated explicitly).

Steve Baird asks about another check. If you have an incomplete type that is tagged, there is a check that the completion is tagged. When does this check happen? Tucker replies at the point that you check the completion. There is some question whether the proposal says that. Steve notes that that means that this check does *not* happen if you never use the bad completion. Randy claims that the wording does say that.

Tucker suggests adding "In the case of a type stub, these checks are performed no later than when a use requires the completion to be available" at the end of 3.10.1(4). Steve Baird is happy that a compiler has the option of flagging it earlier.

Steve Baird comments that he doesn't like the idea that the type has multiple first subtypes. Tucker objects, saying that the incomplete type is really a different type, not a subtype of the completion.

Tucker is asked about the implementation that his compiler team did of this feature. He says that it is incomplete (pun?), as it doesn't handle the fancier type matching rules.

Steve Baird writes a question:

```
type T is ...; -- Incomplete
type A is access T;
O : A;

type T (D : Integer) is ...; -- Completion

... A.all.D –- Legal?
```

Do we want this to be legal? If so, then we're looking at the completion. Tucker points out the last rule of (5-10): in this context, they are the same type.

Pascal queries whether this allows completion with a private type. Tucker answers that it does; that functionality is provided by changing full_type_declaration to type_declaration in the wording.

Erhard wonders what happens if two type stubs meet? Pascal responds that the idea is that it is a lexical check. Tucker points to the last sentence of 3.10.1(11). Steve Baird wonders if "considered" is good RM wording. Tucker shows that this wording is used elsewhere in the manual (for instance, in the accessibility rules). Randy comments that we certainly want to copy the crystal clear accessibility rules, which draws general laughter.

Steve Baird notes that in the last tagged rule (where 'Class is allowed, the last bullet of the second set of bullets in the replacement for 3.10.1(5-10)), "above" is unclear. The old wording used "here", which hardly seems better. Randy points out that this was a single list of bullets before (thus "here" made sense); now there are two lists of bullets.

Several suggestions for fixing the wording are floated. Erhard suggests replacing "above" by "for incomplete types". Tucker suggests changing this "above" to "here". The group finally decides to change "above" to "tagged incomplete types". We return to "considered", and decided to change it after all. Change "considered" in the last paragraph of 3.10.1(10) to "is defined to be".

Someone claims that the third bullet of the last bulleted list of 3.10.1(10) is redundant.

There is a long discussion on these rules. Erhard suggests that there is a problem, as these can be used in access to subprograms (see regular incomplete rules). The second bullet of the same list must state that E has to be either tagged or complete.

Tucker suggests changing “use” to “construct” in the new sentence (added earlier) in 3.10.1(4). (Because the sentence isn’t as clear as it could be.)

Tucker goes on to suggest changing the nested list of three bullets in the last list of three bullets in 3.10.1(5-10). Change the first bullet to:

- E is the same as the completion of D; or
- E is tagged and the same as D; or

He further suggests that we should change “same” to “covers” here.

In 3.10.1(11), change “are considered” to “are defined to be”.

Approve AI with changes: 8-0-1

AI-224 - Pragma Unsuppress

Randy explains the inheritance arguments (see the appendix of the AI for details).

Erhard argues that the inheritance is bad, because you can’t move code around.

Pascal & Erhard argue that whatever the rule is, it should be the same for configuration pragmas as for regular ones.

Steve Baird asks us to consider subunits: A has a non-configuration Suppress pragma; A.B has a configuration Unsuppress pragma. Now consider A.B.C — are checks suppressed? Implementation-defined is ugly here.

We take a series of straw polls:

In favor of inheritance everywhere inside passes easily: 7-0-1.

In favor of configuration pragmas being equivalent to the same pragma occurring as first line of each unit: 7-0-1.

Tucker comments that implementations that don’t inherit Suppress probably don’t need to inherit Unsuppress either. (Because you can’t tell — if the check was not suppressed in the first place, Unsuppress would have no effect).

Someone asks about generics: are you allowed to suppress inside the instance? Tucker claims that an instance body is a copy, so it does inherit from the containing unit. Steve Baird disagrees with this analysis. No conclusion is reached on this point.

Steve Baird notes that there is a similar problem for pragma Inline. What if a routine is inlined in an area where Suppress applies, but Suppress does not apply on the original? Tucker says that this problem is worse if Unsuppress applies inside the inlined routine.

So if Unsuppress applies to the subprogram body, it has to win even if the body is inlined in a region where checks are suppressed. In the absence of an Unsuppress in the body, it is implementation-defined whether a Suppress at the point of call applies.

It appears that generics should follow similar rules.

Tucker would like an explicit permission that it is OK to suppress in inlined code.

Approve intent of AI: 6-0-2

AI-248 - Directory Operations

Jean Pierre explains his proposal. He would prefer we table this proposal, and let him make a working group to do this and other possible related services.

John mentions that perfect is the enemy of good enough. Ada lacks APIs because often we don't have a perfect interface.

Tucker agrees with that, and would like to make this available sooner rather than later.

Randy and Jean-Pierre engage in a technical discussion of the merits of Jean-Pierre's proposal and the AI-248 proposal.

John comments that he would hate to lose momentum on this one API.

Steve Michell suggests that if Jean-Pierre makes a new proposal, he should make it as similar as possible.

Tucker asks how to make progress on this?

The options are to vote it out (approve and forward to WG9), vote to approve and hold, or hold it.

Pascal asks why the proposal has both Rename and Copy_File? Because Rename works on directories.

Approve AI as written and send it to WG9: 8-0-1.

AI-249 - Ravenscar Profile for High-Integrity Systems

Alan Burns give a report on the IRTAW review of AI-249. The open issues are the 5 points enumerated below. There is general agreement that these are the only issues. We will discuss them individually.

Point 1: The minutes of the Bloomington ARG meeting requests that the AI be split into two. But there seems to be three parts:

a) Definition of new restriction identifiers.

b) Definition of pragma Profile.

c) Definition of Ravenscar profile identifier.

Should (a) be one AI and (b)+(c) the other?

Discussion: One of the new AIs is the new restriction identifiers; the other is pragma Profile and the definition of the Ravenscar profile identifier. The intent is that a given profile simply specifies a set of existing pragmas and has no new semantics of its own.

The new restrictions should be put into Annex D; some restrictions currently defined in Annex H should be moved to annex D.

Point 2: Should the Ravenscar profile enforce FIFO_Within_Priorities and Ceiling_Locking, or have these as defaults or allow them to be fixed by the profile_argument_definition (i.e. parameters to the profile)? IRTAW11 voted strongly for enforcement.

Discussion: The group agrees with the IRTAW position.

Point 3: Drop Max_Asynchronous_Select_Nesting => 0 as it is covered by the new identifier No_Select_Statements.

Discussion: The group agrees with this.

Point 4: Section 1.4 of the AI requires that potentially blocking operations be detected. Should this be represented by a new restriction? What identifier should be used? Detect_Blocking?

Discussion: Randy (and others) comment that this doesn't seem like a restriction, more like a sort of policy. Tucker suggests pragma Bounded_Error_Policy (Detect_Blocking). Pascal thinks that this sounds too complex.

Alan comments that “Immediate_Reclaimation” is not really a restriction either. The group groans, then decides to make the new one a restriction. Someone proposes the name No_Undetected_Blocking. This sounds ugly. Erhard suggests No_Nested_Protected_Actions; Alan argues that nesting isn’t an issue. Steve Michell suggests No_Undetected_Blocking_In_Protected_Operation. This is too long. Jean-Pierre suggest No_Blocking as the name. This gives the wrong impression, as the issue is the detection of blocking, not blocking itself. Tucker says it’s the wrong kind of thing, we should stick with “Detect_Blocking”. Exhaustion sets in and the discussion fades away without a firm conclusion.

Steve Michell suggests making it a stand-alone pragma “Detect_Blocking”.

We take a straw poll, counting those in favor of a restriction. This is defeated 3-7-1.

Jean-Pierre asks a general question: What if there is a conflict in pragmas? (Given a pragma Profile and an explicit Restrictions pragma, for example). Pragma Profile is equivalent to a basket of pragmas; whatever rule would apply to a conflict between pragmas would apply here. In particular, if Detect_Blocking is given together with the restriction No_Exception, the program is erroneous if blocking happens.

Point 5. There seems to be some confusion over No_Task_Termination. IRTAW11 understood that this restriction was dropped by the ARG (since it could not be enforced). The workshop therefore felt that normal Ada semantics should apply. Minutes of the Bloomington ARG meeting imply support for No_Task_Termination; its use being to allow an implementation to say what happens if termination actually takes place.

Discussion: No, this item was retained by the ARG. This essentially is a case where each group convinced the other of its original position. The group determines to retain the restriction identifier.

Approve intent of AI: 10-0-0

Alan Burns will update the AI (including the RM wording, identifying the clauses where the wording changes will appear in the RM).

AI-251 - Abstract Interfaces to provide Multiple Inheritance

Steve Baird says he views abstract interfaces as providing a mapping from slot number to slot number. Tucker asks if this is a user view. This gets general laughter. Steve responds that the reason for his view is that derivation doesn’t change anything.

John notes that the discussion section of the AI needs to be updated to match the wording.

Steve continues that the key change in the model is that an abstract interface type *is* an abstract tagged type. (It used to be similar but not the same.)

An abstract interface type is either defined by an abstract type declaration, or derived from an abstract interface type and is abstract.

Pascal would like **interface** in the derived type declaration when defining an interface.

The suggestion is to replace **abstract** by **interface** in the derived type declaration, then adding a legality rule. For example:

```
type T is interface ...

type T2 is new interface T [and N]; -- Bad (see below).

type T3 is new T [and T2] with null record;

type T4 is abstract new T [and T3] with null record;
```

We cannot use the syntax of T2, because that would cause a parsing conflict if **interface** is not reserved (as currently proposed).

So the ARG's legendary creativity for syntax is tapped. Many suggestions are made for deriving from an interface:

```
type T2 is interface new T [and N];

type T2 is interface based on T [and N];

type T2 is interface of T [and N];

type T2 is interface with T [and N];
```

We look at regular tagged extensions to see if any ideas appear:

```
type NT is [abstract] new T with [interface_sequence and]
    [record...end record | private | null record]

interface_sequence ::= subtype_mark |
                       interface_sequence and subtype_mark
```

This syntax can only create a tagged type (either regular or abstract). Interfaces cannot be created with this syntax.

The proposed syntax for interfaces is:

```
type T4 is [limited] interface [with interface_sequence];
```

So, if we inherit from a single interface, we get:

```
type T2 is interface with T;
```

So we don't even need derived interface types, and thus we don't need syntax for them.

Does eliminating derived interface types cause a contract model problem? No, if we derive from an interface type with the regular extension syntax, then we get an abstract tagged type. Erhard still doesn’t believe that there isn't a problem. He asks about the following:

```
generic
    type T1 is abstract tagged private;
package P is
    type T is abstract new T1 with null record;
end P;

type A is interface ...
package PP is new P (A);
```

A doesn't match T1, so there isn't a problem. Several people think we need an interface generic formal type. Steve Baird says no, there are problems with this corner of the language anyway. You don’t know the set of operations involved, so how do we implement it? We don’t want more of this.

Some people do want more of this. Someone says that we should fix the problems in the existing language, not ignore them. It is suggested that using formal interfaces for layering would make sense, where a generic takes an interface, mixes in new operations, then exports a new interface. Tucker shows an example:

```
generic
    type Formal_Interface is interface;
    with procedure Display (X : Formal_Interface’Class);
package G is
    type My_T is abstract new Controlled with
                              Formal_Interface and private;
```

This example seems valuable and likely. Steve Baird finally gives in.

Steve Baird brings up the rule that a completion cannot add interfaces. This seems unfriendly. But without this rule, the contract model and privateness are hosed. Is there any alternative? Steve says no. In the absence of a better solution we have to stick to this restriction.

What about the rules that say one routine may fulfill more than interface? There is a rule that the routine in the two interfaces must be fully conformant. The alternatives are very ugly and heavy.

Someone objects to the changes in section 8. Please do not change the rules for programs that don't use interfaces. These were very difficult to get right, and we don't want to make any accidental changes.

Someone finds the wording "simultaneously inherits" to be obscure. Can we get rid of this wording? Steve Baird says that this means as part of a single declaration.

A suggestion is made to remove "abstract" from terminology, these are just "interfaces".

Tucker would like to see null procedures winning over abstract procedures if there is a conflict. The group does not have a strong opinion on this issue.

Erhard asks why do we define null procedures? Tucker explains the rationale again (it's explained in the AI).

Steve Baird asks about the rule that "A limited interface cannot be implemented by a non-limited type." This rule is necessary because replicated (template) generics would not statically know whether a return type is return by reference. Here is a case where that is no longer known at compile-time:

```
type Lint is limited interface;

function F (X : Lint) return Lint'Class is abstract;
function F2 (X : Lint) return Lint is abstract;
procedure P (X : Lint);

type NLT is interface with Lint;

function F (X: NLT) return NLT'Class;
function F2(X : NLT) return NLT;
procedure P(X : NLT);

type Acc is Lint'Class;
Ptr : Acc;

X : Lint'Class := F (Ptr.all);

P (F2 (P.all));
```

When you call F in a dispatching call, you do not know statically if the result is return-by-reference. This is new; in order to prevent it, we require everything to be limited.

Tucker muses if we could just disallow a return type of the interface type for primitive functions. That would be a contract model problem; a check would be needed at instantiation time. So, we must respecify limitedness each time, so it is explicit. Steve and Tuck will investigate this issue.

Steve Baird will take another stab at this AI.

AI-262 - Access to private units in the private part

Randy describes the change to the rules. Essentially, he enumerated the places where it can appear, because we need to allow nested units and children.

We discuss the wording. Someone asks about private parts of tasks and protected objects. We could just say private part. So reword to:

"…shall appear only within a private descendant of the unit on which the with_clause appears, or within a private part, or body, or pragma in context_clause." Delete the use clause sentence.

The group prefers bullets:

"A name denoting a declaration mentioned only in a with_clause [which]{that} includes the reserved word private shall appear only within:

- a private part;
- a body;
- a private descendant of the unit on which the with_clause appears;
- a pragma within a context_clause."

Tucker is concerned that we don't limit this to the scope of the with_clause. He suggests:

"Within the scope of a with_clause that includes the reserved word **private**, a name denoting a library item mentioned only in that with_clause shall appear only within:

- a private part;
- a body;
- a private descendant of the unit on which the with_clause appears;
- a pragma within a context_clause."

Steve Baird objects. What if there are two private withs for the same unit? This rule seems to not apply.

Reword the rule again:

"A name denoting a library item that is visible only due to being mentioned in with_clauses that include the reserved word private shall appear only within:

- a private part;
- a body;
- a private descendant of the unit on which one of these with_clauses appear;
- a pragma within a context clause."

An example is given:

```
private with PWP;
package Parent is
  …
end Parent;

private with PWP;
private package Parent.Child is
    X : PWP.T;
private
    Y : PWP.T;
end Parent.Child;
```

We need to allow both X and Y here. It appears this wording does that.

Should we retain the rule that allows private withs only on packages? The argument in favor of allowing it anywhere is that it is orthogonal. It doesn't complicate the wording. And it can be useful for automatically generated code. We take a straw poll to allow them everywhere: 7-0-1.

Erhard notes that the summary needs to be fixed to reflect the wording.

Approve AI with changes: 8-0-0.

AI-265 - Partition elaboration policy for High-Integrity Systems

Alan reports on IRTAW discussion on this policy. The conclusion was that it works if the Ravenscar restrictions are imposed, but the group is worried if it works in the whole language. So IRTAW believes it would be OK if this policy is only allowed if Ravenscar is in effect.

Tucker objects, saying that that breaks the equivalence of Profiles to independent pragmas. He asks what restrictions are needed to make this work in the full language.

Alan suggests that No_Task_Allocators and No_Task_Hierarchy would be sufficient.

Tucker wonders about the need for restricting allocated tasks. He doesn't believe that they would be a problem; tasks from allocators can be held along with all of the other tasks.

Jean-Pierre objects that then the task activation list would have to be global. After discussion, it is determined that is OK, as it logically is at the **begin** of the environment task. So it is already global.

Steve Michell points out that a task nested in a function could be called from a package specification. With the tasks being held, how would the function work? Ugh. The group decides that this policy does need to depend on No_Task_Hierarchy; then this example would be illegal.

Jean-Pierre notes that task creation is potentially blocking, and that must be allowed. Alan immediately responds that that had been noted previously, and will be included in the next version of the AI wording (task creation should not be erroneous).

Exceptions and finalization rules are unchanged by this policy.

Pascal summaries the changes: This pragma is illegal unless pragma Restriction (No_Task_Hierarchy) if given in the parition. This is a post-compilation check.

Jean-Pierre comments that he doesn't like the name of the pragma and policy. This brings a groan from the group. ”We’ve discussed this for hours and hours and hours…” The group agrees to avoid discussing it again.

Approve intent of AI: 8-0-2

Alan Burns will update the AI.

AI-266-01 and AI-266-02 - Task Termination procedure

Alan explains the new proposal. The feeling of the IRTAW meeting was that the main additional functionality was grouping of tasks. But that doesn’t buy a lot if you have the full language available. In a restricted environment, you need a simple, restricted proposal.

Tucker thinks that being able to do a group of these is necessary.

Randy wonders if this is useful at all. There are other ways to do this within the language (at least three have been proposed), doesn’t solve many additional problems.

A straw polling on keeping the proposal alive passes easily (9-1-1).

Steve Baird wonders if there is any difficulty telling the different termination conditions apart? The group thinks not, but a discussion of what happens when you are abort while in a handler ensues. It seems that handlers should be abort-deferred.

The new proposal has both regular and protected handlers. Do we want both, or just the protected one? Erhard notes that the problem with the regular handler is that you will need to access global state. So you still need some protection, probably using a protected object. A straw poll on having both kinds of handler fails (0-8-3), so we keep only the protected handlers.

Someone asks why the default parameter is “Current_Task”? It is better to make the default ID be Null_Task_ID, having that mean that it applies to all child tasks of the current task. (This gives us some grouping capabilities.)

The new proposal uses replacement semantics, allowing only one such handler. It is noted that such a limit is always a problem. What if you have two reusable components, both setting termination handlers for the environment task? You do not want the elaboration order to determine what happens (and which component works as designed). Pascal suggests that it could be made a linked list. Alternatively, we could have a check to prevent replacement. But the current proposal would have the components fail to work for no apparent reason.

Another possibility to handle this would be to support an Exchange_Handlers facility, which would let the user chain them.

Someone wonders why all of the concern about handlers for tasks that are not known to the component handling the termination. Is it useful to find out about a task that you don’t know about? Many people think so.

Jean-Pierre suggests changing the difficulty from Medium to High. He jokingly suggests changing the priority from High to Low. Some people seriously agree with that.

Tucker notes that this is another mechanism for hooking events (as was timing events, and of course interrupt handlers). He wonders if we need to make a more general mechanism to handle all of these rather than designing a slightly different one each time.

Alan will revise alternative 02; alternative 01 is left alone for now.

AI-284 - Non-reserved keywords

Tucker says that this is an enabling technology.

Jean-Pierre finds this more surprising than new reserved words.

Pascal says that in practice, there is no difference between non-reserved and reserved keywords. You don’t tell students about the difference, you just give them the whole list.

This isn't a new idea with Ada; other languages have these. Moreover, words like Integer and Standard are very much like unreserved keywords, because using them for something else is ugly.

Tucker suggests that the presentation in the standard should have all of the keywords in one table. A separate rule can explain that the new ones aren’t reserved.

Someone asks why did this get dropped from Ada 9X?

Randy recalls that we had two new reserved words, and four new unreserved keywords in Ada 9X. But there wasn't much benefit for portability; if you have any new reserved words, you have the upward compatibility issue. So we got rid of unreserved keywords. For the amendment, we’re trying for no new reserved words (so we don't have the compatibility issue).

Approve AI with changes: 8-1-0

The opposed vote (Erhard) was because he believes that it would screw up implementation technology. Asked for more details, he explains that sifter technology can do reserved word lookup in six instructions. He claims that it

wouldn't work for unreserved keywords. Randy and Tucker wonder why it wouldn't work; both of their compilers do lexical lookups of unreserved identifiers (Randy gives attribute names as an example).

AI-285 - Latin-9 and Ada.Characters.Handling

Pascal has not yet written this AI. He would like to know whether there is a need for 32-bit characters in Ada? He believes that adding Wide_Wide_xxx is a lot of work. And some implementers report that there isn't much demand for 16-bit characters in Ada.

Tucker suggests to “reserve” the names Wide_Wide_xxx for this, and to allow implementations to support it (much like Long_Long_Integer). This approach gets general agreement.

AI-286 - Assert pragma

Tucker explains the changes. The assertion policy was added (as discussed at the last meeting), and the AI was split into two.

Jean-Pierre thinks that the *Ignore* policy seems wrong. It should not say the pragma is ignored, but that it has no effect. The expression must be legal in any case, it just isn’t evaluated.

Someone wonders why is there procedure Raise_Assertion_Error? It doesn’t seem to buy anything, unless there is a way to provide a replacement body for this routine. Steve Baird points out that a replacement body is problematic: What if the provided routine doesn’t raise an exception? The optimizer would want to assume it does raise an exception. The consensus is to drop the routine Raise_Assertion_Error.

Jean-Pierre does not understand the value of the *Evaluate_and_Assert_True* policy? Pascal explains that if there are side effects, you need to evaluate the assertions so the behavior of the program doesn't change. Steve Baird notes that for expressions that can be proven to have no side effects, the entire thing can be removed.

Steve Michell notes that expressions really should not have side effects, if you are doing static analysis. Tucker comments that this would easily be handled by declaring your own policy.

Erhard does not like the Assertion_Policy. In particular, he does not like *Evaluate_and_Assume_True* and *Assume_True*, as these are unsafe. The group notes that we discussed this extensively last time, and this was the only way to reach agreement. Erhard continues by noting that *Assume_True* effectively changes Assert into Assume. The group agrees with that. These policies are unsafe, but not more than pragma Suppress.

Tucker notes that an implementation can treat the policies *Check* and *Evaluate_and_Assume_True* as the same (if the check fails in the latter case, the program is erroneous, in which case, anything can be done -- including raising Assertion_Error). Similarly, the policies *Ignore* and *Assume_True* can be implemented the same way (no code being generated). Only an implementation that wants to need do anything special with the *Assume_True* variants.

Steve Michell proposes to add “Unchecked” into these two policies: *Unchecked_Assume_True*. The group believes that *Assume_True* is enough.

Alan suggests that the pragma should be Assume, then the policy could be *Assert*. This isn't acceptable, because we’re trying to standardize existing practice.

Steve Michell would like to see an attempt to avoid side effects. That is felt to be impossible, because we need to allow “benign side effects”. And those cannot be defined.

Steve Michell asks Steve Baird about the existing practice with respect to side effects. Steve answers that they (Rational) have seen code where Assert was used solely for side-effects (with a comment to that effect).

Erhard proposes that this exception be moved to Ada.Exceptions. There was not much support in the group for that.

He wonders if there is an implicit dependence on Ada.Exceptions and Ada.Assertions from an Assert pragma. It would be needed because Assert (implicitly) calls Ada.Exceptions and uses the exception in Ada.Assertions. But then pragma Assert cannot be used in a Pure or Preelaborated package (as Ada.Exceptions is not Pure or Preelaborated). So we have to say that it does *not* create a dependence on Ada.Exceptions. A dependence on Ada.Assertions is fine, and should be created.

The package Ada.Assertions is Pure.

Summary of changes: Eliminate procedure Raise_Assertion_Error; make the package pure; allow the exception to be a rename of a non-language defined exception; create no dependence on Ada.Exceptions; and create a dependence on Ada.Assertions.

Approve intent of AI: 6-1-3

Why the negative vote? Two reasons: the policy issue, and the renaming allowance (for which the rationale is "Dewar said so").

Steve Michell comments that we need to codify common practice, but he'll hold his nose voting for it.

Tucker will revise the AI.

AI-288 - Pre/post conditions & Invariants

Tuck describes various issues with the AI.

Any explicit precondition must include the class-wide precondition. One way is to automatically do so (this proposal), or could require programmer to do it (by "and"ing together to provide checks).

Someone wonders if this is available in C++ or Java? No, Java has recently added "assert", but doesn't go further. This is mostly from Eiffel. Tucker believes this is a place where we should be ahead.

Steve Michell has three problems with this proposal.

The first occurs when you are trying to state preconditions/postconditions. Usually you cannot see everything you need to see; so you have to make functions — but these are real functions, not static ones.

The second problem is that you don't have visibility that you need at the point of the pragma.

The third problem is that these aren't task safe. Tucker disagrees. Steve says that you may depend on global state information that is not task safe. Static evaluation assumes all values are evaluated at once; that doesn't happen with dynamic evaluation here.

Jean-Pierre comments that while you are evaluating precondition/postconditions, you need to disable other pre/postconditions. He gives an example of postconditions: Is_Full → not Is_Empty; Is_Empty → not Is_Full. Tucker argues these are not postconditions. Steve Michell agrees, saying that postconditions of a function is usually "True". A correct example would be Is_Full → 'Result = not Is_Empty; Is_Empty → 'Result = not Is_Full.

Still Jean-Pierre's original point is still valid. And that is the rule in Eiffel. To do this properly, it would have to apply to anything that is called. But that is nearly unimplementable (you would need additional checks in every subprogram). Nobody would pay that overhead.

Tucker muses that we might only want to suppress these for function postconditions. If a function postcondition is not inside the function, then it would not be generated. But this rule would apply statically. Otherwise, you would need task-level state to disable checking of postconditions. That doesn't sound good.

Jean-Pierre wonders why 'Incoming makes a copy? Tucker notes that it is only allowed on elementary type.

Erhard asks what happens with the postcondition if there is an exception? Tucker says that exceptional returns do not cause invariant or post checks.

Erhard notes that a precondition can always be written as an Assert in the body, same for postcondition (but it is harder). So, should these be tied to the specification? The answer is mainly because there is only one exception. Essentially, you are hiding the bug; for fault tolerance, everything is an Assertion_Error, and you have to figure it out from the message. If it is being used is strictly for debugging support, it doesn't matter as much.

Alan thinks that static analysis is more important. But this proposal seems to have started at the dynamic end and hopes to work for static analysis. That isn't likely to work. He seconds Steve's concern about visibility. He also notes that static analysis needs things that are not in the program. Tucker says that perhaps you could have a pragma to mark things that exist only for the pre/postconditions and invariants.

John believes that global annotations are the most valuable part of SPARK (and we're not addressing these here).

Erhard comments that postconditions essentially provide a single exit point. We could use Assert if we had a way to insure a single exit point. Perhaps we should have a "Finally" or something like that to provide a single exit point.

Someone wonders why this is on the specification? Tucker says that this is additional specification on the specification. Pascal thinks that this would clutter the specifications and harm readability. Randy concurs, and adds that it also requires a lot of extra mechanism (special visibility, new attributes).

Erhard replies that he would like to allow 'Incoming to occur anywhere. Randy points out that then you would have to copy the parameters every time (which is expensive), or scan the body for uses before generating any code. Tucker agrees that this would be a problem.

If we put these in the body, Postcondition would go right before the **begin** in the declarative part.

In Summary:

We have an unsolved problem with infinite recursion.

We have an unsolved problem with state for static analysis.

John would prefer different names so they don't look like static analysis. Perhaps: "Pre_Check" and "Post_Check" (or "Pre_Assertion" and "Post_Assertions"). He would like to avoid a conflict with "classical static analysis".

Pascal comments that we have two sets of problems, and we can't solve the static analysis issues here. That seems more like a specialized needs annex.

Tucker says that his goal is to enable tools here so that it is portable to move between static and dynamic analysis. Pre and postconditions are a contract, which is why it is valuable to have it on the specification.

John makes a global remark. If we are trying to make it look like Ada is ahead (in this or any other area), we shouldn't use pragmas, rather we want syntax. Tucker replies that if we went with syntax, you would need to put in a lot more information.

We move on the Invariant proposal. Tucker describes the proposal. It describes that a specific property is "usually" true for objects of the type. There are places that they have to be allowed to be False.

Steve Michell again thinks that visibility is an issue; what about child units? Tucker replies that invariants would probably be imposed only on visible parts of visible children.

Jean-Pierre notes that you could have an elaboration order problem, if the function called things in child packages. Tucker and Pascal both say that would be weird. But it would be OK, as it is no worse than any normal elaboration problem.

Erhard asks if the type invariant pragma covers implicit calls (like Adjust and Finalize). Tucker replies that yes, that does not seem to be a problem. It would apply to any user-defined routine called explicitly or implicitly.

Steve Michell suggests that the name of the pragma should be "Invariant_Check" to show the dynamic nature of the check. There is not much support from the group for this.

Type_Invariant really only makes sense if you cannot modify the individual components. It is suggested that this pragma should be defined only for private types. Jean-Pierre says that we should check **in** parameters if it is allowed on non-private types. The group does not like that.

Jean-Pierre notes that this includes Unchecked_Conversion. Yes, that is intentional.

Steve Baird wonders if this includes derived subprograms. Are invariants inherited for derived subprograms? Tucker, after some thinking and discussion decides that for untagged types, it is inherited, and you can't override it. Steve says, "I was going to complain whatever choice you made."

Randy asks, seconded by Pascal if we should split this AI again, as there is much more positive feeling for invariants.

Someone wonders if it should be required to override a type invariant for a tagged type. If not, the invariant is applied only to the ancestor part. So it seems like it should be cumulative, rather than overriding. This is like a postcondition anyway, so it has to be strengthened. For Type_Invariant, calls are statically bound to the parent type's routines anyway. (Classwide_Type_Invariant is dynamically bound.)

Pascal wonders why we need Package_Invariant? Tucker replies that a package could be an abstract state machine. To use Type_Invariant, you'd have to declare a dummy type, and pass it to all of the operations; that seems like we'd be forcing a design on users.

Finally the group responds to Randy and Pascal's call to split the AI again (into pre/postcondition and invariant parts). Pascal would like to get HRG input on static analysis issues. Tucker asks for examples of the information needed for static analysis.

Summary for invariants:

For untagged types, you can define, but not redefine an invariant. This means you may have to create a wrapper for routines that you inherit from your parent. Otherwise, the implementation would have to recompile the whole body, (because it possibly makes calls to other inherited routines), and that is impractical.

For tagged types, invariants are cumulative. New invariants need to be checked.

Tucker will split the AI, and revise both parts.

AI-290 - Declaring functions pure

Tucker explains the proposal. This is essentially an assertion that the function is pure. The rule for pure packages is that you can omit the call and use the previous value. This proposal uses the same rule.

Pascal would prefer a check. Tucker replies that you cannot tell (practically) whether or not there is any problem.

Erhard doesn't understand the rule. If the function changes a variable, can we assume the result is the same? Tucker replies that that is the idea. Erhard says that he would prefer that the arguments are required to be the same, without the side effect permission.

Tucker writes an example to illustrate the issue:

```
if Is_Prime (G) then ...

if Is_Interesting (Q) then ... -- Body of Is_Interesting changes G.

if Is_Prime (G) then ...
```

The result of the first call to Is_Prime can be reused, because we are not looking at the side effects of Is_Interesting.

Erhard likes this rule even less after seeing the example.

John points out that this is a change to the existing wording of the standard.

No one is in favor of this change. This permission is dropped; we're using the 10.2.1(18) wording.

John jokingly suggests that this should be called pragma Pureish.

Steve Baird notes that implementations don't need to do anything with this pragma unless they want to.

Steve Baird suggests that the pragma should be Assume_Pure. Randy quickly notes that we're codifying existing practice, so changing the name isn't an option.

Do we want this AI? A vote to keep it alive passes 7-1-2.

The negative vote was cast because he doesn't like the non-semantics preserving semantics.

Tucker will provide wording and changes to the AI.

AI-292 - Sockets Operations

We have a brief discussion on the best way to proceed to get this API into the standard. The consensus is that we probably ought to start with AdaSockets (it is the most portable of the existing interfaces and is reasonably abstract).

Randy takes an action item to make an effort to start a group to create a sockets API. It is recommended that he not write the proposal.

AI-293 - Built-in hash function

Jean-Pierre would be interested in this proposal. It would make things more portable.

Erhard doesn't believe that this would be useful. For instance, for names, default implementations assume randomly distributed keys, which never happen.

Pascal points out that it is very easy to create a general hash function with a stream. Steve Baird says that it is easy to write a generic unit to do this.

Pascal objects to this capability being defined as an attribute. This would require us to duplicate the stream attributes mess (rules for composition, etc.). He also does not think that we would ever agree on a hash that was good enough.

Tucker thinks that we should add a generic hash package that could be implemented with streams. There is value in providing a portable package for this.

Steve Baird would like to insure that we don't define it to depend on 'Address of the object; it must depend on the contents.

The generic should take a generic private type, with a result type that is a formal discrete type.

Tucker will write the AI.

AI-296 - Vector and matrix operations

John recommends that these operations be put into the Numerics annex. There is general agreement with this proposal.

Pascal suggesting separating the accuracy rules from rest. He also suggests cleaning up and simplifying the language of the proposal submitted by the UK.

John will take the AI.

We vote to put the operations in annex G: 8-0-0.

AI-297 - Timing events

Alan reports that IRTAW gave the highest priority to the following issues: Ravenscar, time budgeting, timing events, and user scheduling. (Proposals will be coming on time budgeting and user scheduling.)

Turning to the time events proposal. The problem with the existing language (that is, using delays) is that you need to write tasks whose only job is to adjust the priorities of other tasks. There is an example of this given in the IRTAW11 paper, which can be found in the appendix of the AI.

The proposal is to provide *time handlers*, which work similarly to interrupt handlers.

Tucker queries why there is a timing event object. Alan responds that it is needed so you can identify the event for canceling, etc. There is also an implementation benefit, as the implementation might want to put it on chains, etc. rather than using implicit memory allocation.

Jean-Pierre wonders why it isn't possible to simply attach a procedure to a hardware clock (interrupt). Alan responds that doesn't work because we need to use the run-time system's clock. Different clocks would not solve the problem. Jean-Pierre is not convinced. He doesn't believe that the need is that important, because he thinks most people will use a hardware clock. Pascal does not agree, saying that an implementation cannot allow attaching to the run-time's clock, because it has a bunch of requirements that cannot be violated.

Pascal queries why this package is so different from the existing interrupts package? Alan responds that this is not a periodic item, it only fires once. Interrupts differ by being permanently attached.

Tucker believes that if this is a one-shot, the event object should be passed to the handler so that it can be reenabled if needed. That means that Parameterless_Handler is passed the timing event object. (If this is adopted, better change the name of the handler type!) There is some support for this idea.

Randy takes Tucker's suggestion further, suggesting that the event object should be declared to be tagged. Then it is possible to add extension components, such as the period, in the object. He notes that extensions would be optional; it would still be possible to use the object without an extension. The object probably will need to be controlled (or an equivalent) anyway, so that it can properly be detached if it is finalized while still attached to a handler.

Steve Michell suggests that you could parameterize by adding discriminants to the protected type of the handler. So adding extension components doesn't buy much.

Tucker comments that he would like to see additional queries. As it is, we have a private type that essentially holds one bit of information. Jean-Pierre suggests that we could add a query for the time span value.

Tucker believes that Cancel_Handler should return the existing handler; otherwise we have a race condition with Is_Handler_Set. (It would take two operations to cancel a handler, and the state could change between them.)

Alan summaries the changes: We need to add the handler to Cancel_Handler. Add a Current_Handler function. Add the event object parameter to Parameterless_Handler (and change name of the handler type).

Do we need an Exchange_Handler routine? We take a straw poll, and we decide that we don't as the idea is defeated 1-6-4.

Steve Baird says that he doesn't like a "must be library level rule". He suggests that if you made the parameters a general access type, then we get the library level check for free (as part of the accessibility check).

Randy points out that we already have such a rule for interrupts (C.3.1(8)). But Steve counters that that is a contract violation. After discussion, the group concludes that he is correct. Ouch. A long discussion ensues about this existing language problem.

Tucker suggests making the “must be library-level rule” a run-time rule. It usually could be checked at compile-time, but may have to be done at run-time in some implementations in generics. This solution was used for accessibility, which is a similar problem.

The group reaches a consensus that we need to fix interrupt handlers the same way as time handlers.

Three choices have been identified:

- Use access parameters, letting the accessibility rules do the work.
- Detect it (at runtime potentially).
- Leave it erroneous.

The first choice cannot be used for existing interrupt handlers.

Thus the group decides that it must be detected, possibly at run-time. This is Ada, after all, leaving it erroneous is not appealing.

The editor is directed to open an AI to fix C.3.1(8). Use a solution similar to accessibility rules for this check.

Alan Burns will revise the AI.

AI-298 - Non-preemptive scheduling

Alan explains the proposal. It defines new policies for non-preemptive scheduling. It also defines a “token”, an execution resource (as in D.2.1). You cannot lose the token, only give it away. Tucker wonders why the proposal doesn't just say “processor”. Alan replies that you can’t overturn the basic semantics, which is always required to be preemptive.

Jean-Pierre notes that interrupts are still allowed. This means that there still must be a lock on a protected objects used as interrupt handlers. Alan agrees that is correct.

Tucker thinks that this proposals looks like an addition to a house where you can see what is the original house and what is the addition years later. He would rather rewrite D.2.1 completely to make this possible. Alan believes that much more extensive changes would be needed.

Steve Baird wonders if the definition of “sequential” needed to be changed. Alan says that it does not. It might be too conservative, but that is OK.

Jean-Pierre is concerned that the definition of Non_Preemptive_Locking is inconsistent. After discussion, he is convinced that it is correct as written.

The question is raised whether pragma Priority should be disallowed in protected objects in this mode, or just ignored. Disallowing it is safer (users will get what they expect or an error). But disallowing it is not as portable, because code would be made illegal just by the presence of the pragma.

Steve Michell claims that this is a fairly significant change to the semantics. He believes that it will change the scheduling of next task. Tucker disputes that. Tasks are queued at the old priority, and otherwise you are non-preemptive.

Tucker asks Alan if the purpose is to eliminate the ceiling checks. Alan says it is. But you would still have to check for calls in Interrupt_Priority objects; the ceiling model still applies there.

Erhard would like to get rid of the post-compilation rule. He thinks that it should be implied. His argument is that it is annoying for the user: the binder will give a message saying that “you must have so-and-so pragma” in your program. If the implementation knows that, why didn't it give it? Alan replies that he is being consistent with the current language. Moreover, it is possible to use Non_Preemptive_FIFO_Within_Priorities without Non_Preemptive_Locking; the post-compilation check only occurs when Non_Preemptive_Locking is used.

Tucker muses that this maybe should be Suppress (Ceiling_Check); it doesn't have anything to do with locking. In fact, D.3(14) lets you get rid of these priorities altogether. Therefore, we don't need a separate locking policy, just have this implied by Non_Preemptive_FIFO_Within_Priorities.

Alan agrees with this analysis, and will redo the AI that way.

Tucker would like a vote on changing the wording to get rid of this token thing. The group doesn't want to vote without knowing the magnitude of the change. Tucker agrees to let Alan show him some of the places that would need to be changed.

Alan will revise the AI.

AI-299 - Defaults for generic formal parameters

Pascal argues that having such default just makes the code more obscure.

Tucker counters that we have defaults for other things, why should these be limited to just objects and subprograms?

Tucker notes that **use** is a bad choice, because you could leave out a semicolon and it would still be legal syntax. We tried hard to avoid that in Ada.

We could invent a new keyword ("default" is suggested). Another suggestion (using existing keywords) is **until others**.

The generic renaming proposal can be implemented with a generic skin package. It also appears very complex and messy to implement. We take a straw vote on dropping the generic renames part of the proposal, which passes easily: 6-0-2. This effectively means "No action" on part 3 of the proposal.

Pascal wonders if we should we support <> defaults on this? John replies that <> defaults are a mess, we don't want to expand this further.

Tucker notes that it is odd that **in out** would allow a default here, and not in subprograms. Randy comments that we could fix subprograms, too. There is no support for that from the group. However, Pascal notes that **in out** formal objects are really renames, the syntax is misleading.

We take a straw vote on keeping the AI alive. It passes, 4-2-2.

Tuck presents an idea he has for partial parameter lists for generic formal packages. See details below.

Randy will update the AI. He asks about the syntax, and is told to come up with something. (Thanks for the help; what happened to that famous ARG syntax creativity?)

AI-300 - The standard storage pool

This proposal is thought to be too complicated. Pascal objects to this because it doesn't fit their implementation well.

The default policy in the Rational compiler is complicated because there isn't a global storage pool. To get a pool, you have to declare an access type at a precise location, and the characteristics of the type will determine those of the pool (in particular, its lifetime). So it's not like there is a one-size-fits-all storage pool.

Tucker suggests that we simply declare a global object for a global storage pool. Define it to be just a global pool. But what type would that object have?

Steve Baird claims that

```
type Acc is access …;
The_Pool : Storage_Pool'Class renames Acc'Storage_Pool;
```

is an adequate idiom. It feels more like a workaround to some.

The ARG votes this proposal no action: 7-0-1.

AI-301 - Missing operations in Ada.Strings.Unbounded

Randy explains that the readability of programs using unbounded strings is a problem, because you have to convert to type String to do anything interesting.

Jean-Pierre comments that unbounded strings are really for storage; don't use them for manipulation. That doesn't seem to be the intent expressed in the standard.

Tucker would like to see a procedure version of To_Unbounded_String. He also would like to add a defaulted starting parameter to all the Index functions. Pascal immediately claims that that is not compatible.

Tucker hates making this look like an add-on. The new parameter would have to be at the end. In that case, only renames (and overriding via derivation) would be incompatible. These are unlikely.

There is not much interest in the slice version of the operations that were proposed.

The group feels that I/O is generally valuable. Complex has this, and it is a child of Text_IO. But you need access to the representation of unbounded string. So it appears that it has to be a child of Unbounded. Steve Baird objects, you could have an implementation package as a child of Unbounded.

Thus, we settle on the name Ada.Text_IO.Unbounded_IO to make it like Complex_IO. It could be a rename from an implementation package. There also would be a wide version (Ada.Wide_Text_IO.Unbounded_IO), of course.

The From parameter will need to be added to all index functions (for Fixed, Bounded, and Unbounded).

Randy will update the AI.

AI-302 - Data structure components for Ada

Randy explains that this AI is a placeholder for whatever proposals will come in this area.

Tucker says that we should reply to Jeff (the proposer), encouraging him to get involved with the other groups working on this area.

Jean-Pierre asks what the workshop (held at the same time as the ARG meeting) decided.

Tucker replied that he had attended part of their meeting. They decided not to use the Booch components. They were working on maps and vectors (growable arrays). They were working using the signature idea (see the discussion in the next item), so multiple implementations are possible.

Partial parameter lists for generic formal packages

Tucker describes another problem in using generics. That is the all-or-nothing approach for parameters to formal packages. He says that the data structures people are going to run into this, as they are planning heavy use of signature packages.

An example of the problem they will face:

```
generic
    type Key is private;
    type Elem is private;
    type Map is private;
```

```
    with procedure Put (M : in out Map;
                        K : in Key;
                        E : in Elem) is <>;
    with function Get (M : in Map;
                       K : in Key) return Elem is <>;
package Map_Signature is end;

generic
    type Key is private;
    with function Hash ...
    type Elem is private;
package Hash_Map is
    type Map is ...
    procedure Put ...
    function Get ...

    package Hash_Map_Sig is new Map_Signature (Key, Elem, Map);
end Hash_Map;
```

A use of the signature would be:

```
generic
   type Key is private;
   with package A_Vector_Sig new Vector_Sig (<>);
   with package A_Map_Sig new Map_Sig (<>) –- No;
                                           -- have to use (Elem, Key, Map);
package ...
```

But you really want:

```
generic
   type Key is private;
   with package A_Vector_Sig new Vector_Sig (Key, <>);
   with package A_Map_Sig new Map_Sig (Key, <>)
package ...
```

Erhard explains that the problem is that you cannot reference formal parameters of a signature package in another signature package.

The group sees interest in this idea, and asks Tucker to make a proposal.

Detailed Review of Regular AIs

AI-276 - Interfaces.C.Strings.Chars_Ptr_Array has aliased components

Remove “(but it’s still gratuitous junk).”

Approve AI with change: 6-0-1.

AI-294 - Instantiating with abstract operations

Randy has already written this up. We will discuss this later. (Not at this meeting, as it turned out.)

AI-295 - Another violation of constrained access subtypes

Discussion of the three options for fixing this problem:

- Assume the worst rule in the body;
- Run-time check;
- Make general access subtypes illegal.

We discuss the third option (removing constraints on general access subtypes). This would allow eliminating the requirement that aliased components (and objects) are constrained. Then this problem and others like it don’t occur. Steve Baird says that this is very incompatible, because the constrained item can be much smaller than the unconstrained. Tucker replies that objects would be incompatible, but for components, you can already change the discriminant with an aggregate assignment.

Is there some way to check whether this incompatibility is a real issue? Tucker says that we need set up an incompatibility checking infrastructure. Randy and Erhard argue that it would be hard to get source code submitted, and a tool will only work on one platform.

After the meeting, Tucker and Randy discussed this, and Tuck realized that this would be a contract model violation on formal access types (which can match both pool-specific and general access types). So option 3 is out.

Steve Baird will write the AI.