Minutes of the 47th ARG Meeting

15-16 June 2012

Stockholm, Sweden

Attendees: Steve Baird, John Barnes (except Saturday afternoon), Randy Brukardt, Alan Burns (Friday only), Jeff Cousins, Brad Moore (via Skype, all but a couple of hours on Friday), Erhard Ploedereder, Jean-Pierre Rosen, Ed Schonberg, Tucker Taft (except late Saturday afternoon), Tullio Vardanega.

Observers: None.

Meeting Summary

The meeting convened on 15 June 2012 at 14:00 hours and adjourned at 16:30 hours on 16 June 2012. The meeting was held on Friday in a meeting room at Näringslivets Hus and on Saturday in a meeting room at the Mornington Hotel, both in Stockholm, Sweden. The meeting covered most of the agenda.

AI Summary

The following AIs were approved with editorial changes:

AI05-0298-1/01 Last-minute presentation issues in the Standard (9-0-1)
AI12-0022-1/02 Changing the exception raised for an assertion (9-0-0)

The intention of the following AIs were approved but they require a rewrite:

AI12-0003-1/01 Default storage pool for storage pools (7-0-0)
AI12-0026-1/01 Reentrant categorization (7-0-0)
AI12-0028-1/01 Import of variadic C functions (9-0-0)
AI12-0030-1/01 Formal derived types and attribute availability (7-0-2)
AI12-0031-1/01 All_Calls_Remote and indirect calls (8-0-2)
AI12-0032-1/02 Questions on 'Old (10-0-0)
AI12-0034-1/01 Remote stream attribute calls (6-0-5)
AI12-0035-1/01 Accessibility checks for indefinite elements of containers (10-0-0)

The following AIs were discussed and assigned to an editor:

AI12-0002-1/02 RCI units should not allow types with user-defined stream attributes
AI12-0016-1/00 Implementation model of dynamic accessibility checking
AI12-0020-1/01 'Image for all types
AI12-0023-1/01 Make Root_Stream_Type an interface
AI12-0027-1/01 Contract violation for aliased components of actuals for formal array types
AI12-0033-1/01 Sets of CPUs when defining dispatching

The following AIs were discussed and put on hold:

AI12-0024-1/01 Compile-time detection of range and length errors
AI12-0025-1/01 Allow 'Unchecked_Access on subprograms

[Note: it is premature to rule out any ideas for the next revision of Ada. The ideas associated with these AIs are currently not expected to be included in future versions of Ada, but that will be revisited when we formally start work on such a revision. Note that the following AI is a special case, as we believe the problem is solved as a consequence of another feature, so it does not need to be considered further.]

The following AI was discussed and voted No Action:

AI12-0029-1/01 Relax requirement for functions to have return statements (8-0-0)Detailed Minutes

Previous Meeting Minutes

Randy applied all of the previously reported changes to the minutes (this is the currently posted "draft version 3").

Approve minutes as posted: 10-0-0.

Date and Venue of the Next Meeting

Next meeting: Boston, December 6-9, 2012, in association with the HILT 2012 conference [the new name of SIGAda's conference]. The conference schedule is not finalized yet; in particular, we don't know if there will be sessions on the afternoon of Thursday, December 6th, . If not, we'll start our meeting then and finish up on the afternoon of the 8th; if there are sessions on Thursday afternoon, then we'll meet the afternoon of the 7th through the morning of the 9th. WG 9 will meet Friday morning (7th), we won't meet then in any case.

Next year's Ada-Europe meeting will be in Berlin, Germany; we'll expect to meet afterwards as usual.

Ada 2012 Rationale

John Barnes reports that he is finishing up the 4th chapter for publication soon in the Ada User Journal. He expects to have the remaining chapters finished by spring 2013.

Randy Brukardt reports that the ARA has sponsored the creation of HTML and text versions of the Rationale; the first three chapters are finished and are awaiting review and some corrections by John. John promised to do the review and corrections by the end of the month (June).

ASIS

WG 9 has instructed us to restart work on the ASIS standard, working closely with implementers.

We have seen no interest from implementers in the semantic subsystem. As such, we need to plan to remove that from the Standard.

Jean-Pierre will take an action item to do two things:

- look for unnecessary implementation freedom in the existing ASIS standard draft; and
- find as many places as possible where critical functionality was left to the semantic interface so that we can consider how to add that functionality to the syntactic interface. Both the SIs and the minutes should be examined for this.

Sergey Rybin should be asked for suggestions on open Ada 2012 interfaces (some were worked out in !ASIS sections). [Editor's note: Gary Barnes at IBM also should be involved if possible.]

Thanks

We thank Ada-Europe for the accommodations on Friday. We thank AdaCore for the partial support for the Saturday meeting room. We thank Brad Moore for his amazing fortitude in attending nearly the entire meeting via Skype despite it occuring in the middle of the night (1 am to 8 am) at his location in Calgary, Canada.

Old Action Items

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.

Steve Baird:

- AI12-0016-1 (with Tucker Taft)

- AI12-0020-1
- AI12-0027-1
- AI12-0028-1
- AI12-0032-1
- AI12-0035-1 (with Gary Dismukes)
- Review AI12-0030-1 after Randy Brukardt creates it
- Consider proposing a checked specification that a subprogram is non-blocking (see AI12-0026-1).

Randy Brukardt:

- AI12-0030-1 (review by Steve Baird)
- AI12-0034-1

Editorial changes only:

- AI05-0298-1
- AI12-0022-1
- AI12-0029-1

Gary Dismukes:

- AI12-0035-1 (with Steve Baird)

Brad Moore:

- AI12-0023-1
- AI12-0026-1
- AI12-0031-1

Erhard Ploedereder:

- AI12-0003-1

Jean-Pierre Rosen:

- Study the draft ASIS standard for changes that we should make to adapt it to Ada 2012 (see ASIS above).

Ed Schonberg:

- AI12-0002-1

Tucker Taft:

- AI12-0016-1 (with Steve Baird)
- AI12-0033-1
- "Pure is not pure enough for distribution", split from AI12-0031-1

Detailed Review

The minutes for the detailed review of AIs and SIs are divided into ASIS Issues (SIs), Ada 2005 AIs (AI05s), and Ada 2012 AIs (AI12s), but 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 18 of the Ada 2012 AARM. Paragraph numbers in earlier drafts may vary.

Detailed Review of Ada 2005 AIs

AI05-0298-1/01 Last-minute presentation issues in the Standard

There is a typo in the summary, “Preelaborare”.

The subtype in (3) is missing `Integer range`.

Approve AI with changes: 9-0-1.

Detailed Review of Ada 2012 AIs

AI12-0002-1/02 RCI units should not allow types with user-defined stream attributes

The actual question of the AI is easily solved. But the question of what restrictions apply to instances in an RCI unit remains (and might complicate solving the real problem).

After much discussion, Tucker boils it down to whether the subprograms in an instance are remote subprograms and whether they (and the rest of the instance) have to follow the restrictions of the RCI unit.

It seems like the entire model of an RCI unit would collapse if that wasn't true. Tucker says that it could work as if the generic was instantiated as a Remote_Types package, and that would work semantically. But whether that makes any sense in terms of RCI is less clear.

Instantiations should act like nested packages if possible. But if there is a lot of code depending on these being allowed in the single existing implementation of this Annex, perhaps we should consider an alternative to allow instantiations to act differently. We'll need feedback from AdaCore (the source of the original question) on this topic.

Assign to Ed to get feedback and determine the course forward on this one.

AI12-0003-1/01 Default storage pool for storage pools

The problem is that the normal trick:

```
type Foo is access Integer;

type Bar is access Interesting;
for Bar'Storage_Pool use Foo'Storage_Pool;
```

does not work because Foo has no storage pool if Default_Storage_Pool(null) has been specified as a configuration pragma.

Steve thinks that if you say Default_Storage_Pool(null), you really don't want a system-defined pool. He would rather have an aspect that specified that you revert to the default pool-selection behavior for a specific access type (ignoring the Default_Storage_Pool).

Randy notes that another problem that we were trying to solve was passing a pool as a generic parameter. We can do some of that with the aspect Default_Storage_Pool.

Jean-Pierre asks again why we don't have a name for the system storage pool. Steve explains that it isn't a single pool necessarily, it is more a strategy for selecting a storage pool.

Randy notes that the reason you need this is that you might want to use the system pool to implement the user-defined pools, that you then are required to use in the rest of your system. In that case, you probably want a configuration of Default_Storage_Pool(null) in order to ensure no one creates an access type with the system pool. But you still need to be able to access it in the package that defines the pools. This seems like an important case.

If we had System.Default_Storage_Pool, that represented a policy rather than a pool, we would be OK. One suggestion is T'Default_Storage_Pool:

```
       Storage_Pool => T'Default_Storage_Pool;
```

but that probably would be a freezing problem.

Steve suggests an aspect Ignore_Default_Storage_Pool. Randy jokingly expands it to Ignore_Default_Storage_Pool_and_Choose_System_Storage_Pool_Instead.

Use_System_Storage_Pool is suggested. That seems like a reasonable idea.

Erhard suggests adding pragma Standard_Storage_Pool; it works like the default storage pool pragma (including scoping) but selects the Standard_Storage_Pool. That seems to work best.

Randy notes that we'd need an aspect that applied to instances, to be similar to the aspect that is available for the default storage pool.

We could use a special marker in the Default_Storage_Pool pragma, but nothing comes to mind as appropriate.

Erhard will provide wording and a more complete write-up for this AI.

Approve intent: 7-0-0.

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

The problem here is that we neither know how to efficiently implement the rules as currently defined nor how to change the rules so that they can be efficiently implemented. Someone needs to take the lead on that.

The only real alternative is to abandon dynamic accessibility checking altogether, and the increase in erroneous execution that implies is uncomfortable at best.

Tucker and Steve will take this one and try to develop an implementation model (and/or propose rule changes) in order to determine what we need to do.

AI12-0020-1/01 'Image for all types

It would be nice if Image could be user-specified (for all types). Then there should be a default implementation.

What is the image of an access type? Steve thinks that it should be implementation-defined. With the exception that the image of **`null`** is “null”.

Tucker suggests that the image of an access type is always “null”. That's not liked, because it is not reversible.

The format is an aggregate for composite types.

Tucker suggests defining a “text streaming” operation, and then the Image is just defined in terms of that.

Steve suggests that it might be easier to define such a streaming operation because we would then use the existing rules for stream attributes.

Steve will take this AI and create a realistic proposal.

Keep alive: 10-0-0.

Raise the priority of this AI.

The underlying type needs to be Wide_Wide_String. Tucker thinks that is grim. He suggests having all three kinds.

AI12-0022-1/02 Changing the exception raised for an assertion

Erhard asks why this is parenthesized? Because we parenthesized all of the other "statements" as expressions (conditional expressions, quantified expressions).

Erhard thinks that **or else raise** would be an idiom.

Tucker wonders if we had a Raise_Exception function, we wouldn't need any syntactic changes. Randy notes that would require the 'Identity on the exception name, which is clunky. Tucker agrees and withdraws his suggestion.

Erhard gives an example showing that we don't want to require parentheses on these:

```
Pre => ((X /= null) or else raise Constraint_Error)
        and then ((Mode = In_File) or else raise Mode_Error with "xxx"))
```

Erhard prefers a "nice precondition" to a "nice expression".

Oh-oh: there is an ambiguity if there is a with clause here:

```
raise Mode_Error with "xxx" & "foo" -- ambiguous.
```

Does the & associate with **raise** or "&"??

Similarly:

```
"A" & raise Constraint_Error with "B" & "C"
```

So we have to require parens if the with clause is present. It could be optional otherwise.

It is noted that this looks better in conditional expressions without the parentheses:

```
(if P then raise Program_Error)
```

taking advantage of the implicit `else True`. (And this is probably easier to understand than the "idiom" using **or else**.)

```
::= raise name
  | (raise name with expression)
```

Erhard argues that it should not be "any type". Jean-Pierre points to the example in the AI as to why it is valuable to allow any type.

Tucker notes that `A + raise` is weird. He would like to make this low precedence to force parentheses unless it is the "only thing".

Make this into a relation, so it has to stand pretty much alone or be in parentheses. We won't add this to choice_relation, as it doesn't make much sense to use this in a case statement or membership operation.

If you have

```
(raise Constraint_Error) or (raise Program_Error)
```

it is unspecified which is raised (as the order of evaluation is unspecified). If you write something silly, you get silly results – no surprise there.

Steve notes that

```
return raise Unimplemented;
```

satisfies the rules for requiring a return statement and works for any type.

Since we've solved the ambiguity problem with precedence, we don't need any parentheses at all in the syntax.

Specifically, 4.4(3) gets an additional line relation ::= raise_expression

Drop the wording about parentheses.

11.3 title should be “Raise Statements and Raise Expressions”

In 11.3(3), the “if any” feels confusing. Randy notes that we could put Raise_Expression first, but then we ought to do that consistently.

> The name in a raise_expression or in a raise_statement, if present, shall denote...

Tucker suggests that we just say

> “An *exception*_name shall denote ...”
>
> “A *string*_expression is expected to be of type String.”

Steve notes that there are other *String*_expressions in the Standard.

> “A s*tring*_expression of a raise_statement or raise_expression is expected to be of type String.”
>
> “An *exception*_name of a raise_statement or raise_expression shall denote...”

Steve notes that 11.2 never requires that *exception*_name denotes an exception. So apparently the following is legal:

```
exception
   when Integer | Program_Error => -- hope not
```

So, add before 11.2(6), “An *exception*_name of an exception_choice shall denote an exception.”

Brad notes a typo: “it it resolves” in the discussion after function Foo.

Steve would like “raise expression” to be mentioned in the subject.

Approve AI with changes: 9-0-0.

AI12-0023-1/01 Make Root_Stream_Type an interface

Brad tries to explain his last proposal. The idea was that multiple inheritance problems were prevented by not allowing the adding of any further interface to any type that might have a hidden interface.

The idea is that we assume that you've derived privately from a hidden interface for any private type, so a hidden interface cannot be added to any extension of a private type, unless that private type says that it didn't use any hidden interface.

Randy then explains his scheme (based on an idea of Tucker's). Private types can specify that they allow interfaces in the private part. Such types would not allow any interfaces to be added to extensions.

Ed says that both of these seem like a lot of implementation work for little gain, so he is against both.

Randy notes that there isn't a good solution to create a controlled stream type. It is suggested that a controlled component be used with a self-referential access discriminant. Randy says that proves his point about “no good solution”. :-)

Randy notes that the incompatibility here is unlikely to occur in practice. Perhaps it would make sense to simply take the incompatibility and make streams an interface. It is unlikely to make sense to hide the streams. Not everyone agrees with this position. [Editor: I recall Tucker(?) giving some example where it might make sense to hide streaming, but I didn't write down any details.]

Randy notes that he had also considered a model where the operations of a hideable interface were restricted, such that everything has to be overriding. Steve and Tucker say that model would make problems in extensions, because they could clobber behavior that is expected by inherited routines (other than those defined by the interface, which would have to be overridden). It doesn't look like that idea would work.

Erhard suggests that we refactor into an interface with an existing abstract type derived from it. But then the interface type still has to be visible, and the incompatibility remains. (This is suggested every time this topic comes up for any existing type.)

We need more compelling examples for this in order to implement either Brad's or Tucker's solution. We have several years before we have to make a formal decision on this, so we will hope for better examples.

Keep AI alive: 7-0-2.

AI12-0024-1/01 Compile-time detection of range and length errors

Randy notes that some of these cases are easy. But you have to be careful to avoid checking in dead code.

Randy says that there is another option: adding some sort of soft error. The idea is that the error would be detected and the program rejected unless an aspect is set on the code.

Ed notes that this would be incompatible, as aspects would have to be added in various places.

This could be considered a tool quality issue. But it’s hard on users to depend on things that are “tool quality”.

Erhard notes that specifying this in the language means that compilers would only follow the language-defined rules, it would prevent clever compilers from learning more. Randy notes that we could define rules that would allow going further, but that would potentially be a portability problem. Ed notes that a static analysis tool would provide far more information about this than a compiler could.

Steve notes that in the glaringly simple cases, this is bizarre to allow. But we all agree that the problem is the "not as obvious" cases.

AI12-0025-1/01 Allow 'Unchecked_Access on subprograms

The argument in favor is that GNAT has 'Unrestricted_Access and it is widely used, because the accessibility checks get in the way. But there is little interest in the complexity of the proposal. Moreover, Erhard objects to allowing more ways to write erroneous programs.

AI12-0026-1/01 Reentrant categorization

The idea is to follow Pure categorization rules, but allow “safe” variables. This doesn't guarantee safe use by multiple tasks – you can still have race conditions (deadlook, livelock, etc.) if there is more than one variable in the package.

This seems safe and not particularly complex to define and implement. We wonder about generic instances; that should work in the same way it works for Pure.

Steve mentions that he would like to have a specification that a subprogram promises that it is non-blocking (that is, it does not call a potentially blocking operation). That probably would go into a separate AI. Erhard launches into a claim that this rule (potentially blocking) is unneeded in the first place. The editor did not record his discussion because he didn't understand it at all. Steve will send a proposal to Ada-Comment if he wants to discuss this idea further (it seems interesting).

Returning to reentrant. This seems useful.

Erhard complains that “reentrant” is too broad; if he has a normal package that is wrapped by a reentrant package (say by using locks), that could not have the categorization. So he suggests a different term, perhaps “monitored”.

Someone looks on-line and reports on the definitions of these terms. "Monitored" implements reentrant operations with encapsulation. Reentrant without encapsulation is "normally" reentrant.

The categorization is incompatible with restriction No_Reentrancy (H.4(23)). We wonder about No_Recursion; it probably is only a problem if No_Reentrancy is set (which we already decided that it could not be).

Approve intent: 7-0-0.

Brad gets this AI by default (it was his idea).

AI12-0027-1/01 Contract violation for aliased components of actuals for formal array types

Tucker wonders whether we even care about the back conversion (which is what we are discussing here). The problem is passing an array of unaliased components to a parameter that takes an array of aliased components. The back conversion is harmless, because nothing bad can happen. (Outside accesses to the components know whether the components are aliased, of course.)

We could even repeal 4.6(8) for this issue, but that's pretty complex.

So a Ramification would be sufficient.

One possible problem occurs with formal subprograms that have parameters of a formal array type. Randy is dubious that there is a problem. Tucker says that there is a real problem that he thinks we need to fix.

Steve worries that an 'Unchecked_Access might be pointing at an object that doesn't exist (if the array conversion was by-copy) [using a formal subprogram]. The problem only exists in this generic case because otherwise the back conversion makes it illegal. But this requires a formal array type, formal subprogram, and a structural array conversion of a global variable declared outside of the generic (or passed as a parameter to the generic). Not very likely.

Steve worries that this is more likely than we think. If so, we have to go back to one of the solutions (2) or (3).

Steve is going take this AI and create some examples to convince us of a problem, and define an assume-the-worst rule to detect it.

Steve mentions something about non-view conversions of by-reference arrays. Randy thinks that was previously handled somehow. Steve will try to prove this one way or another.

Keep AI alive: 9-0-0.

AI12-0028-1/01 Import of variadic C functions

Steve proposes that we have a family of conventions that includes the number of fixed arguments. We need that information to be part of the conventions.

Tucker asks what the number ought to be. It's the number of fixed arguments. So something which is all varargs would use C_Varargs_0.

Erhard wonders what it means to support this on Exported items. That surely will work, but the routine still will have a fixed number of parameters, so C callers would have to provide the correct number (which is weird).

So Import is required, Export is implementation-defined. What about Convention for access-to-subprogram? It seems necessary with interfacing, but it doesn't seem to be useful for calls.

Tucker thinks that we don't need access-to-subprogram, you can always use a wrapper that is an Ada routine that makes the call to the original routine. [Editor's note: This seems dubious if the access-to-subprogram is in the C interface and the Ada code needs to create values to pass to the C routines.]

So this also should be implementation-defined.

It seems useful to require detection if there are too few parameters for the number.

Steve wonders if Varargs is the right name, or if we want to use some other name.

Tucker looks in the C standard, and finds that “varying” is the term. That doesn't make any sense.

We like “variadic”. Wikipedia documents that, and in fact “Varargs” just redirects to “Variadic”. C_Variadic_0, C_Variadic_1, etc.

The maximum N is implementation-defined.

Steve will create wording for this one.

Erhard is not convinced it is worth it. He thinks it is big mechanism, and most targets don't need it. But it is pointed out that the target with this problem is x64 -- on Linux, Mac, and Windows. That's where 90% of new programming is going to be. And we want portability between implementations on those targets.

[90% is a way-over-the-top number here. It might be true just considering traditional PCs/servers, but there is a lot more to computing than just that. It probably doesn't apply to iPad/iPhone/ianything or Android or Windows RT/Windows Phone. However, it was the number mentioned during the discussion, so it remains here - Editor]

Approve intent of AI: 9-0-0.

AI12-0029-1/01 Relax requirement for functions to have return statements

There is no need for a solution to the problem per se, because **`return raise`** `Unimplemented;` (as defined in AI12-0022-1, previously approved) works fine for this problem.

There is no interest in a more general solution.

So we're done here. The problem was solved differently. Add some text to that effect to the AI.

No action: 8-0-0.

AI12-0030-1/01 Formal derived types and attribute availability

The example should have “with Ada.Streams”.

The subject should include “stream attribute availability” since that's what's defined (other attributes don't have availability).

Write this up as a Ramification and add a To Be Honest note to explain. We won't make any wording changes since it is widely expected to work this way and the changes would be very likely to break something else.

The best solution would be to make availability a characteristic, but 7.3.1 defines how characteristics are inherited for private types while 13.13.2 defines how availability is determined for private types – these aren't quite the same.

Otherwise, we have to write a bunch of special wording specifically for this case. That will be tough to get right (look at how many iterations of 12.5.1(20-21) we've already written).

Steve wonders where the dynamic semantics come from. That follows from the inherited characteristics; that says that the ancestor's stream attribute (and availability) is inherited and that's what would be called.

Randy will write the To Be Honest note and ask Steve to review the result.

Approve intent: 7-0-2.

AI12-0031-1/01 All_Calls_Remote and indirect calls

Tucker says that a call though an access-to-subprogram value should always be considered as being outside of the declarative region.

Tucker suggests that “direct” be added to the two existing sentences of E.2.3(19/3).

He then suggests that all calls through a remote access-to-subprogram with All_Calls_Remote set then are remote calls, even if on the same partition. Otherwise, calls to the same partition can be local (at the choice of the implementer).

What is All_Calls_Remote good for? Jean-Pierre says that it is useful for testing a PCS.

Anonymous access is never remote. But those are illegal for remote subprograms (because they never have external streams).

Normal access-to-subprogram is always a local call, because it can only be a non-visible type, only visible to a package or its children. These are all in the (same) declarative region, so there is no problem.

Brad will take this AI and create wording.

Ouch – a Pure package can declare a normal access-to-subprogram type. A shared passive package could declare a global variable. So you could make a normal call that crosses partitions. But that's not supposed to happen. All of the other avenues are blocked (no variables in Remote type or RCI), no access-to-subprogram in Shared_Passive.

We need to somehow block Shared_Passive from declaring such things. Tucker says that the problem is broader than that: Shared_Passive assumes that no access types can be imported from Pure packages. Something needs to be fixed.

Tucker will take this one, split into a new AI, because this is a much broader problem. “Pure is not pure enough”.

Approve intent of (existing) AI: 8-0-2.

AI12-0032-1/02 Questions on 'Old

Jean-Pierre suggests 3 levels of expectations on how 'Old could work:

1) X = X'Old (= predefined)

2) For any F where F(X)=F(X), then F(X'Old) = F(X)

3) F(X'Old)=F(X)'Old

3) seems too complicated and doesn't work when there are side-effects.

1) seems like a user expectation that we should require.

2) isn't as clear; it seems like it should be required.

We look at Steve's last proposed wording. Randy doesn't believe that the type of the constant is really as specified if the tag is different.

What about a named number? For instance, PI'Old. The problem is that there aren't constants of type universal real. Steve suggests making static expressions illegal here.

But that's not enough. Tucker notes that X.A'Length'Old has type universal integer. That seems like a useful feature.

So we probably just want to say that the type might be universal. We already have rules for universal evaluation at runtime. Probably an AARM note is enough.

The AI will definitely need to be a binding interpretation.

Approve intent: 10-0-0.

This AI stays with Steve Baird.

AI12-0033-1/01 Sets of CPUs when defining dispatching

How the defined numbering corresponds to the underlying system is implementation-defined. It's not necessarily a direct mapping.

Randy notes that an implementation has no practical way to find out the best numbering for a particular user, when it might be controlled by a particular implementation of Linux. The number and organization of CPUs is likely to be determined at runtime; it's likely that programs are compiled to run on any appropriate target, anything from a single core processor to sets of multithreaded multicore processors. The appropriate CPU mappings may vary wildly.

The set implementation is looking more interesting. Randy notes that a lot of operations would need to be changed.

Tucker wonders if it makes more sense to support a function to let the users update the mapping between the target's underlying numbering and the numbers that the CPU functions use.

Steve notes that if a task is running then, things would be weird. This remapping operation could only be executed when no other tasks (other than the environment task) are running.

What happens if you refer to CPUs beyond Number_of_CPUs? It seems like such values shouldn't be allowed here (it's not possible to run a task using a CPU over that value – D.16(14/3)).

After a break, Tucker says that he notes that there is already a problem in that the System_Dispatching_Domain might have holes. For example, if the program is running on a system with 8 CPUs, and a domain is Created containing CPUs 3 through 6, the System_Dispatching_Domain holds the remaining CPUs, which in this case are CPUs 1, 2, 7, and 8. Then Get_First_CPU(System_Dispatching_Domain) = 1 and Get_First_CPU(System_Dispatching_Domain) = 8. but this surely does not completely characterize the values in the System_Dispatching_Domain. Moreover, you can create many holes in the System_Dispatching_Domain this way.

So he thinks that it makes more sense to add set operations here in order to properly represent any of these items. And that fixes the problem that was the original question.

Tucker volunteers to write up a solution on this basis.

Steve notes that Assign_Task does not work on a child task that inherits a non-system domain from its parent. The aspect can be specified (even to System Dispatching Domain), but not Assign_Task. That seems weird.

Keep AI alive: 9-0-0.

AI12-0034-1/01 Remote stream attribute calls

The problem here really is that stream attributes always come along everywhere.

Tucker wonders if we would be better off making it illegal to have a remote access-to-class-wide of a (specific) type that has user-defined stream attributes. In that case, the attributes are not available for the type (which must be limited), and then they cannot be called this way.

Randy worries about the incompatibility. This seems narrow enough, and requires enough hoops to jump through to do it, that it doesn't seem like a major problem.

Randy will propose wording for this one.

Approve intent of AI: 6-0-5.

AI12-0035-1/01 Accessibility checks for indefinite elements of containers

We need to somehow bring the allocator checks into the container semantics. Steve suggests defining “creation of an element” and define which operations do this. "Creation of an element" would then do the “checks associated with an initialized allocator”.

Steve comments that if these checks are suppressed, and a “natural” implementation is made, the checks are suppressed. That generally isn't allowed for the containers, and it doesn't seem to make sense here, either.

Can this occur in definite containers? Probably not, because a mutable nonlimited type cannot have access discriminants. Still, it would be easier to define this for all containers rather than trying to determine when it can or cannot happen.

Steve will work with Gary to propose wording for this AI.

Approve intent: 10-0-0.